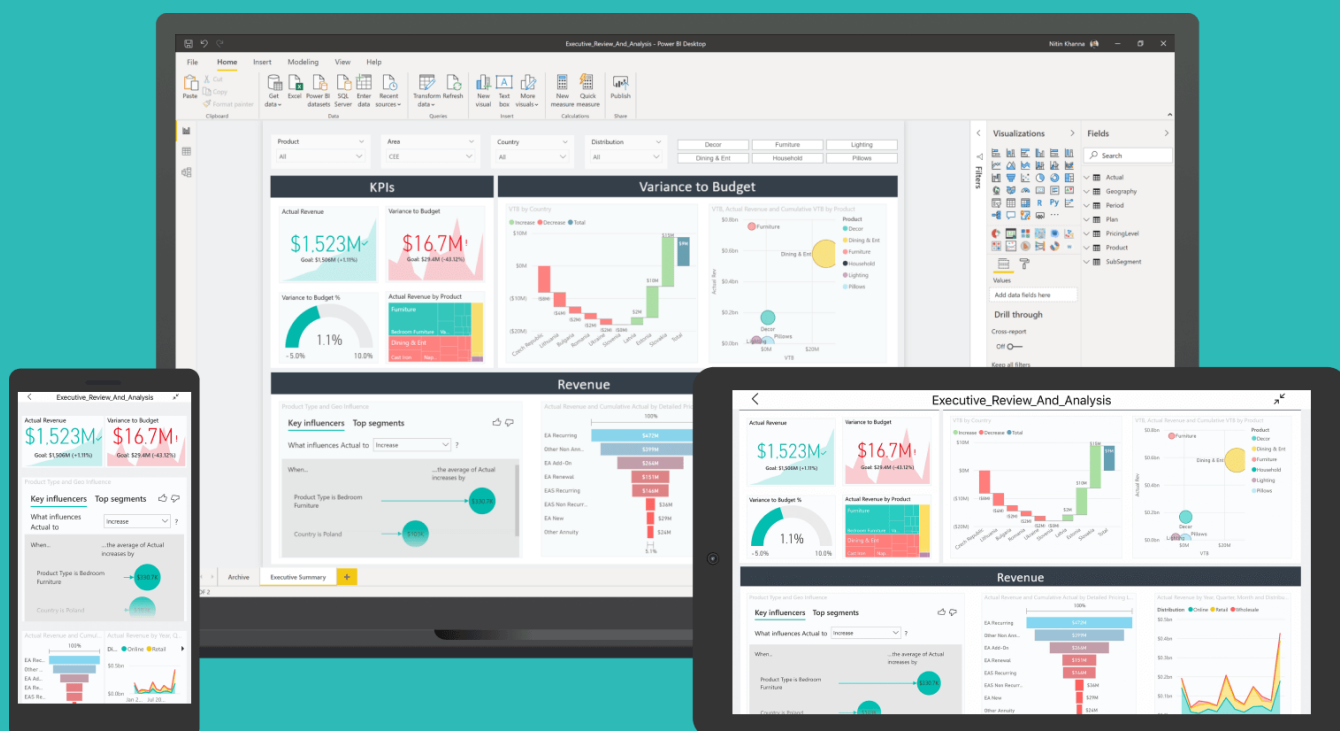


А. Д. ОБУХОВ, И. Л. КОРОБОВА

АНАЛИЗ И ОБРАБОТКА ИНФОРМАЦИИ В ОФИСНЫХ И ОБЛАЧНЫХ ТЕХНОЛОГИях



Тамбов
Издательский центр ФГБОУ ВО «ТГТУ»
2020

Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Тамбовский государственный технический университет»

А. Д. ОБУХОВ, И. Л. КОРОВА

АНАЛИЗ И ОБРАБОТКА ИНФОРМАЦИИ В ОФИСНЫХ И ОБЛАЧНЫХ ТЕХНОЛОГИЯХ

Утверждено Учёным советом университета в качестве учебного пособия
для студентов и магистрантов направлений подготовки
09.03.01, 09.04.01 «Информатика и вычислительная техника»
дневной формы обучения

Учебное электронное издание



Тамбов
Издательский центр ФГБОУ ВО «ТГТУ»
2020

УДК 004.9
ББК з973.53
О-26

Рецензенты:

Кандидат педагогических наук, доцент, доцент кафедры математического моделирования и информационных технологий Института математики, физики и информационных технологий ФГБОУ ВО «ТГУ имени Г. Р. Державина»
Е. В. Клыгина

Доктор технических наук, профессор,
проректор по НИД ФГБОУ ВО «ТГТУ»
Д. Ю. Муромцев

Обухов, А. Д.

О-26 Анализ и обработка информации в офисных и облачных технологиях [Электронный ресурс] : учебное пособие / А. Д. Обухов, И. Л. Коробова. – Тамбов : Издательский центр ФГБОУ ВО «ТГТУ», 2020. – 1 электрон. опт. диск (CD-ROM). – Системные требования : ПК не ниже класса Pentium II ; CD-ROM-дисковод ; 33,0 Mb ; RAM ; Windows 95/98/XP ; мышь. – Загл. с экрана.
ISBN 978-5-8265-2174-8

Посвящено решению задач анализа и обработки информации в информационных системах, основанных на облачных и офисных технологиях. Представлены разделы по основам программирования, обработке информации в офисных приложениях и облачных системах и работы с данными на языке Python. Приведены примеры решения задач анализа и обработки табличной информации, визуализации, создания, редактирования и чтения документов, организации сетевых репозиторий и работы с API.

Предназначено для студентов и магистрантов направлений подготовки 09.03.01, 09.04.01 «Информатика и вычислительная техника» дневной формы обучения.

УДК 004.9
ББК з973.53

*Все права на размножение и распространение в любой форме остаются за разработчиком.
Нелегальное копирование и использование данного продукта запрещено.*

ISBN 978-5-8265-2174-8

© Федеральное государственное бюджетное образовательное учреждение высшего образования «Тамбовский государственный технический университет» (ФГБОУ ВО «ТГТУ»), 2020

Развитие современных офисных и облачных технологий неразрывно связано с применением передовых методов и подходов в области интеллектуального анализа и обработки данных. Системы поддержки принятия решений, технологии искусственного интеллекта, методы по обработке и анализу больших данных позволяют повысить качество и гибкость информационных систем.

Для успешной интеграции перечисленных технологий необходимо обладать необходимыми компетенциями в области анализа и обработки данных, уметь правильно пользоваться существующими программными решениями и владеть навыками разработки новых программных средств.

В рамках данного пособия рассматривается задача подготовки специалистов и разработчиков к решению различных прикладных задач с применением офисных и облачных технологий. На первом этапе читатель ознакомится с основами языка программирования Python, обладающего широким набором библиотек по анализу и обработке информации. Эти знания позволят ему решать задачи хранения и анализа больших объемов данных, их первичной обработки и визуализации.

Во второй главе рассматривается автоматизация обработки данных в офисных приложениях и технологиях. Разбираются такие задачи, как автоматизация формирования и редактирования документов, работа с таблицами, документами формата PDF и LaTeX.

Третья глава посвящена обработке информации в облачных технологиях и сервисах. На примере фреймворка Flask будут рассмотрены основы создания простейших веб-приложений, работа с сетевыми запросами, а также взаимодействие с API существующих облачных сервисов.

Данное учебное пособие будет использоваться при подготовке студентов и магистров по следующим направлениям: 09.03.01 – Информатика и вычислительная техника (профиль «Модели, методы и программное обеспечение анализа проектных решений»), дисциплины: «Офисные технологии», «Облачные технологии», «Системы подготовки документации»; 09.04.01 – Информатика и вычислительная техника (магистерская программа «Модели, методы и программное обеспечение анализа проектных решений»), дисциплины: «Интеллектуальные системы», «Методы организации информатизации промышленных систем», «Проектирование информационных систем предприятий».

1. ОСНОВЫ АНАЛИЗА И ОБРАБОТКИ ИНФОРМАЦИИ С ПОМОЩЬЮ ЯЗЫКА ПРОГРАММИРОВАНИЯ PYTHON

1.1. Основные понятия системного анализа и обработки данных в информационных системах

Определим некоторые основные понятия, применяемые при решении задач обработки и анализа данных в информационных системах, облачных и офисных технологиях.

Информация – сведения об объектах, явлениях и событиях, процессах окружающего мира, передаваемые различными способами (устно, письменно, графически), что позволяет уменьшить неопределённость знаний о них.

Информационная система – организационно упорядоченная совокупность информационных объектов и информационных технологий, в том числе и с использованием средств вычислительной техники и связи, реализующих информационные процессы.

Облачные технологии – информационные технологии, обеспечивающие сетевой доступ к некоторому массиву вычислительных ресурсов (серверам, хранилищам, приложениям, сервисам) с возможностью их оперативного предоставления и освобождения.

Обработка информации – комплексный процесс взаимодействия с информацией путём осуществления над ней множества операций (например, ввод, запись, чтение, удаление и т.д.) с помощью некоторого набора программных и технических средств.

Офисные технологии – информационные технологии, направленные на организацию, автоматизацию и оптимизацию деятельности по управлению движением информационных потоков предприятия.

Системный анализ – научный метод, применяемый для установления структурных связей между переменными или постоянными элементами исследуемой системы.

Таким образом, реализация эффективных информационных систем на основе облачных и офисных технологий требует разработки соответствующего программного обеспечения, направленного на решение задач анализа и обработки информации. В качестве инструмента для их решения предлагается использовать язык программирования Python.

1.2. Основы языка программирования Python

Python – это высокоуровневый язык программирования, поддерживающий структурное, объектно-ориентированное, функциональное, императивное и аспектно-ориентированное программирование. Синтаксис Python направлен на получение минималистичных программ с высокой читабельностью программного кода. Основными особенностями языка являются: динамическая типизация, автоматическое управление памятью, полная интроспекция (возможность запросить тип и структуру объекта), механизм обработки исключений, поддержка многопоточных вычислений, высокоуровневые структуры данных [1].

Python поддерживает динамическую типизацию, т.е. тип переменной определяется в момент присваивания ей некоторого значения, хотя правильнее будет обозначить этот процесс как «связывание значения с некоторым именем переменной».

Python является объектно-ориентированным языком программирования. Все типы в Python являются объектами, например:

- bool (булевый: True или False),
- str (строка),
- int (целое число),
- float (вещественное число),
- None (NoneType, «ничего»).

Объектами являются также коллекции: список (упорядоченный набор данных различных типов), кортеж (неизменяемый список), словарь (ассоциативный массив), множество, а также все функции, методы, модули, классы.

Важным моментом при разработке на Python является выделение блоков кода с помощью отступов (пробелов или табуляций) вместо, например, фигурных скобок. Поэтому неправильный отступ приведет либо к синтаксической ошибке, либо нарушит логику работы программы. Правила оформления кода приведены в стандарте PEP8, с которым вы можете ознакомиться на сайтах [2]:

<https://www.python.org/dev/peps/pep-0008/> (официальный сайт)

<https://pep8.ru/doc/pep8/> (русскоязычный перевод).

1.2.1. Основные операторы в Python

Python поддерживает все необходимые арифметические операции [3]: сложение (+), вычитание (-), умножение (*), деление (/), возведение в степень (**), взятие остатка (%), целочисленное деление (//).

Оператор ввода *input()*. Возвращает введенную с клавиатуры строку.

Оператор вывода *print(X)*, где *X* – объект или перечисленная через запятые совокупность объектов различных типов (строки, числа, списки и т.д.).

Для преобразования типа (например, строка в число) используется вызов функций, соответствующих названию нужного типа: *int("5")*, *str(4)*. При этом создаётся новый объект нужного типа.

Условный оператор обозначается *if*, после оператора пишется условие (скобки необязательны) и ставится двоеточие. Внутренний блок имеет отступ. Альтернативный блок может быть размещён после оператора *else*. Если условий и альтернатив несколько, возможно сокращение *elif*.

Пример.

```
a = input()
a = int(a)
if a == 0:
    print("Null")
elif a > 0:
    print("Positive")
else:
    print("Negative")
```

Операторы цикла *while* и *for*. Цикла с постусловием в Python нет. Цикл *for* может перебирать любую последовательность, в том числе символы строки, элементы массивов, списков, множеств и т.д. Для создания счётчика цикла можно использовать команду *range(s, n, step)*, где *n* – верхняя граница счётчика (не входит в перебираемую последовательность), *s* – начальное значение (по умолчанию 0), *step* – шаг изменения счётчика. Шаг может быть отрицательным. Общий вид цикла *for* имеет вид: «*for* счётчик *in* последовательность:».

Внутри цикла возможно применение команд *break* и *continue* для прерывания цикла и перехода сразу к следующей итерации.

Пример.

```
a = 10
while a > 0:
    a = a - 1
for i in range(1, 10, 2):
```



```

print(i)
if i == 7:
    break

```

Оператор определения функции или метода класса *def*. Внутри может содержать оператор *return* (возврат) для возврата объекта из функции или метода. Функция в Python всегда возвращает результат, даже если оператор *return* отсутствует (в этом случае будет возвращен *None* – «ничего»). Функции будут подробно рассмотрены далее.

Оператор *pass* используется для пустых блоков кода.

1.2.2. Работа с коллекциями

Списки – упорядоченные изменяемые коллекции объектов произвольных типов. Объявляются командой *list()* либо с помощью пустых квадратных скобок.

Список можно создать с помощью списочного выражения:

```
a = [i ** 2 for i in range(10)]
```

Списки (как и строки) поддерживают такой инструмент, как срезы – извлечение последовательности элементов. Синтаксис среза: *a[i:j:step]*, понимается как извлечь элементы списка *a*, начиная с *i*-го и заканчивая (не включительно) *j*-м с шагом *step*.

Основные методы списков:

<i>list.append(x)</i>	Добавляет элемент в конец списка
<i>list.extend(Y)</i>	Расширяет список <i>list</i> , добавляя в конец все элементы списка <i>Y</i>
<i>list.insert(i, x)</i>	Вставляет на <i>i</i> -ю позицию значение <i>x</i>
<i>list.remove(x)</i>	Удаляет первый элемент в списке, имеющий значение <i>x</i>
<i>list.pop([i])</i>	Удаляет <i>i</i> -й элемент и возвращает его
<i>list.index(x, [start [, end]])</i>	Возвращает индекс первого элемента со значением <i>x</i> (поиск может проводиться от <i>start</i> до <i>end</i>)
<i>list.count(x)</i>	Возвращает количество элементов со значением <i>x</i>
<i>list.sort([key=φ функция])</i>	Сортирует список на основе заданной функции
<i>list.reverse()</i>	Сортирует список в обратном порядке
<i>list.copy()</i>	Копирует список
<i>list.clear()</i>	Удаляет все элементы списка

Кортежи – это неизменяемые списки. Задаются командой *tuple()* или перечислением элементов в круглых скобках через запятую. Кортежи занимают меньше памяти, чем списки. Кортежи могут использоваться для множественного присваивания, например:

a, b = 1, 2

a, b = b, a

Причем сначала осуществляются вычисления в правой части, а затем их результаты присваиваются в левую часть каждой переменной соответственно. Количество элементов слева и справа должно совпадать.

Строки – упорядоченные последовательности символов, используемые для хранения и представления текстовой информации. Также являются коллекциями.

Рассмотрим основные функции и методы строк:

<i>S + S2</i>	Конкатенация (сложение) строк
<i>S * 5</i>	Повторение строки
<i>len(S)</i>	Длина строки
<i>S.find(str, [start],[end])</i>	Поиск подстроки в строке. Возвращает номер первого вхождения или –1, если элемент не найден
<i>S.replace(шаблон, замена)</i>	Все вхождения «шаблона» меняются на «замены». Третий необязательный аргумент – количество замен
<i>S.split(символ)</i>	Разбиение строки по разделителю
<i>S.upper()</i>	Преобразование строки к верхнему регистру
<i>S.lower()</i>	Преобразование строки к нижнему регистру
<i>S.join(список)</i>	Объединение строки из элементов списка с разделителем <i>S</i>
<i>ord(символ)</i>	Символ в его код ASCII
<i>chr(число)</i>	Код ASCII в символ

Множества – коллекция, содержащая неповторяющиеся элементы в случайном порядке. Могут использоваться при логических вычислениях: объединении или пересечении множеств. Создаются командой *set()*.

Операции над множествами:

<i>len(A)</i>	Число элементов в множестве <i>A</i>
<i>x in A</i>	Принадлежность <i>x</i> множеству <i>A</i>
<i>A.isdisjoint(B)</i>	Истина, если <i>A</i> и <i>B</i> не имеют общих элементов
<i>A == B</i>	Истина, если все элементы <i>A</i> принадлежат <i>B</i> и наоборот
<i>S.issubset(B)</i>	Все элементы <i>A</i> принадлежат <i>B</i>
<i>A.union(B, ...)</i> или <i>A B</i>	Объединение нескольких множеств
<i>A.intersection(B, ...)</i> или <i>A & B</i>	Пересечение нескольких множеств
<i>A.copy()</i> -	Копия множества

Словари – неупорядоченные коллекции произвольных объектов с доступом по ключу (ассоциативные массивы). Создаются с помощью команды *dict()* или пустых фигурных скобок.

Добавление значений в словарь происходит присваиванием значения по ключу:

```
a = {}  
a["test"] = 1  
a[3] = "value"  
a[(1, 0, 1)] = {"k": "v"}
```

Перебрать все элементы словаря можно в цикле:

```
d = {"a": 1, "b": 5, "c": 9, "d": -100}  
for key, value in d.items():  
    print(key, "=", value)  
for item in d:  
    print(d[item])
```

Метод *items()* возвращает пары «ключ-значение» и за счёт множественного присваивания они распределяются в переменные *key* и *value*. Во втором цикле мы перебираем только ключи словаря.

Основные методы словарей:

<code>dict.clear()</code>	Очищает словарь
<code>dict.copy()</code>	Создаёт копию словаря
<code>dict.get(key[, default])</code>	Возвращает значение ключа, если его нет, возвращает <i>default</i> (по умолчанию None)
<code>dict.items()</code>	Возвращает пары (ключ, значение)
<code>dict.keys()</code>	Возвращает ключи словаря
<code>dict.pop(key[, default])</code>	Удаляет ключ и возвращает значение. Если ключа нет, возвращает <i>default</i> (если <i>default</i> не задан – выдаёт исключение)
<code>dict.values()</code>	Возвращает значения в словаре

1.3. Функциональное и объектно-ориентированное программирование в Python

1.3.1. Функции в Python

Функция – объект, принимающий аргументы и возвращающий значение. Сначала необходимо объявить функцию с помощью команды *def*, после чего можно будет осуществлять её вызов. Определим простейшую функцию и вызовем ее:

```
def test(x, y):  
    return x + y
```

```
test(1,3)
```

Функция может быть любой сложности и возвращать любые объекты (списки, кортежи и даже другие функции).

Функция может принимать произвольное количество аргументов или не принимать их вовсе. Также распространены функции с произвольным числом аргументов, функции с позиционными и именованными аргументами, обязательными и необязательными. Рассмотрим все эти случаи.

Например, укажем необязательный аргумент *c* со значением, по умолчанию равным 2: `def func(a, b, c=2)`.

Функция также может принимать переменное количество позиционных аргументов, тогда перед именем ставится `*`: `def func(*args)`. `args` – это кортеж из всех переданных аргументов функции.

Функция может принимать произвольное число именованных аргументов, тогда перед именем ставится `**`: `def func(**kwargs)`. `kwargs` – словарь, где ключи – это имена переменных.

Функция может не иметь имени, быть анонимной. Такие функции называют **lambda**-функциями. В отличие от обычных они не требуют инструкции `return`.

Пример: `func = lambda x, y: x + y`

1.3.2. Работа с классами в Python

Создадим простейший класс:

```
class Test:
    def m1(self, a1, a2):
        pass
```

При создании собственных методов обратите внимание на то, что метод определён внутри класса. Также у методов всегда есть хотя бы один аргумент, который принято называть `self`. В него передаётся тот объект, который вызвал этот метод. Когда программа вызывает метод объекта, Python передаёт ему в первом аргументе экземпляр вызывающего объекта, который всегда связан с параметром `self`. Иными словами, `test.hello_world()` преобразуется в вызов `hello_world(test)`. Этот факт объясняет особую важность параметра `self` и то, почему он всегда должен быть первым в любом методе объекта.

Для имён атрибутов и методов применяются те же правила, что и для имён переменных и функций. Имя должно быть записано в нижнем регистре, слова внутри имени разделяются подчёркиванием.

Технология сокрытия информации о внутреннем устройстве объекта за внешним интерфейсом из методов называется **инкапсуляцией**. Свойство кода работать с разными типами данных называют **полиморфизмом**.

В Python принята «утиная типизация». Название происходит от шуточного выражения «если нечто выглядит как утка, плавает как утка и крякает как утка, то это, вероятно, утка и есть». В программах на Python это означает, что если какой-то объект поддерживает все требуемые от него операции, то с ним и будут работать с помощью этих операций, не заботясь о том, какого он на самом деле типа.

Чтобы полиморфизм корректно работал, за ним надо следить как на уровне синтаксиса (одинаковые имена методов и количество параметров), так и на уровне смысла (методы с одинаковыми именами делают похожие операции, параметры методов имеют тот же смысл).

При работе с объектами бывает необходимо в зависимости от их типа выполнить те или иные операции. С помощью встроенной функции *isinstance(object, type)* мы можем проверить тип объекта. Первый параметр представляет объект, а второй – тип, на принадлежность к которому выполняется проверка. Если объект представляет указанный тип, то функция возвращает True.

Специальные методы имеют для интерпретатора особое значение. Имена специальных методов и их смысл определены создателями языка: создавать новые нельзя, можно только реализовывать существующие. Названия всех специальных методов начинаются и заканчиваются на **два подчёркивания**.

Пример такого метода – `__init__`. Он предназначен для инициализации экземпляров и автоматически вызывается интерпретатором после создания экземпляра объекта.

Остальные специальные методы также вызываются в строго определённых ситуациях. Большинство из них отвечает за реализацию операторов. Так, например, всякий раз, когда интерпретатор встречается запись вида $x + y$, он заменяет её на `x.__add__(y)`, и для реализации сложения достаточно определить в классе экземпляра `x` метод `__add__`.

Другие специальные методы

Метод	Описание
<code>__add__(self, other)</code>	Сложение ($x + y$). Будет вызвано: <code>x.__add__(y)</code>
<code>__sub__(self, other)</code>	Вычитание ($x - y$)
<code>__mul__(self, other)</code>	Умножение ($x * y$)
<code>__truediv__(self, other)</code>	Деление (x / y)
<code>__floordiv__(self, other)</code>	Целочисленное деление ($x // y$)
<code>__mod__(self, other)</code>	Остаток от деления ($x \% y$)
<code>__divmod__(self, other)</code>	Частное и остаток (<code>divmod(x, y)</code>)
<code>__radd__(self, other)</code>	Сложение ($y + x$). Будет вызвано: <code>y.__radd__(x)</code>
<code>__rsub__(self, other)</code>	Вычитание ($y - x$)

Метод	Описание
<code>__lt__ (self, other)</code>	Сравнение ($x < y$). Будет вызвано: $x.$ <code>__lt__</code> (y)
<code>__eq__ (self, other)</code>	Сравнение ($x == y$). Будет вызвано: $x.$ <code>__eq__</code> (y)
<code>__len__ (self)</code>	Возвращение длины объекта
<code>__getitem__ (self, key)</code>	Доступ по индексу (или ключу)
<code>__call__ (self[, args...])</code>	Вызов экземпляра класса как функции

Наследование – это механизм, позволяющий запрограммировать отношение вида «класс В является частным случаем класса А». Наследование является способом повторного использования кода между классами без необходимости нарушения инкапсуляции. Это достигается за счёт того, что производный класс может пользоваться атрибутами и методами базового класса.

Если класс наследован от другого класса, то проверка существования метода (или атрибута) осуществляется так:

- сперва метод ищется в исходном (производном) классе;
- если его там нет, он ищется в базовом классе;
- предыдущие шаги повторяются до тех пор, пока метод не будет найден, или пока процедура не дойдет до класса, который ни от кого не наследуется.

Процедура, когда метод производного класса дополняет аналогичный метод базового класса, называется расширением метода, а в коде это выглядит следующим образом:

```
class Child(Parent):
    def __init__(self, size):
        print('child')
        super().__init__(size, size)
```

Функция `super()` возвращает специальный объект, который делегирует («передаёт») вызовы методов (в данном случае – метода `init`) от производного класса к базовому. Эту функцию можно вызывать в любом методе класса, например, в конструкторе.

Python предоставляет возможность наследоваться сразу от нескольких классов. Такой механизм называется множественным наследованием, и он позволяет вызывать в производном классе методы разных базовых классов.

Хотя в языке и зафиксирован порядок разрешения таких конфликтов (в общем случае классы просматриваются слева направо), эта особенность может привести к ошибкам при использовании множественного наследования.

1.4. Работа с библиотеками в Python

Модулем в Python называется любой файл с программой. Модули могут быть объединены в библиотеки. Каждая программа может импортировать модуль и получить доступ к его классам, функциям и объектам.

Подключить модуль можно с помощью инструкции *import*. После ключевого слова *import* указывается название модуля. Одной инструкцией можно подключить несколько модулей.

```
import os  
import time, random
```

После импортирования модуля его название становится переменной, через которую можно получить доступ к атрибутам модуля. Например, можно обратиться к константе *e*, расположенной в модуле *math*: *math.e*

Стоит отметить, что, если указанный атрибут модуля не будет найден, вызовется исключение *AttributeError*. Если не удастся найти модуль для импортирования, то – *ImportError*.

Если название модуля (а также входящей в его состав функции или переменной) слишком длинное или его требуется изменить по другим причинам, то для него можно создать псевдоним с помощью ключевого слова *as*.

```
import math as m
```

Теперь доступ ко всем атрибутам модуля *math* осуществляется только с помощью переменной *m*, а переменной *math* в этой программе уже не будет.

Подключить определённые атрибуты модуля можно с помощью инструкции *from*. Она имеет несколько форматов:

```
from math import e as E  
from math import *
```

Первый формат позволяет подключить из модуля только указанные вами компоненты. Для длинных имён также можно назначить псевдоним, указав его

после ключевого слова *as*. Импортируемые атрибуты можно разместить на нескольких строках для лучшей читаемости кода.

Второй формат инструкции *from* позволяет подключить все переменные из модуля.

Следует заметить, что не все атрибуты будут импортированы. Если в модуле определена переменная `__all__` (список атрибутов, которые могут быть подключены), то будут подключены только атрибуты из этого списка. Если переменная `__all__` не определена, то будут подключены все атрибуты, не начинающиеся с нижнего подчёркивания. Кроме того, необходимо учитывать, что импортирование всех атрибутов из модуля может нарушить пространство имён главной программы, так как переменные, имеющие одинаковые имена, будут перезаписаны.

1.4.1. Создание модуля

Создадим файл `test.py`, в котором определим какие-нибудь функции:

```
def hello():  
    print('Hello, world!')  
  
def test_function(n):  
    return n ** 2
```

Теперь в этой же папке создадим другой файл, например, `main.py`:

```
import test  
  
print(test.hello())  
print(test_function(2))
```

Модуль нельзя именовать также, как и ключевое слово. Также имена модулей нельзя начинать с цифры. И не стоит называть модуль также, как какую-либо из встроенных функций – это приведёт к тому, что встроенный модуль будет замещён и перестанет быть доступным. То же самое относится и к названиям функций.

Модуль можно использовать как самостоятельную программу, однако, при импортировании модуля его код выполняется полностью. Этого можно избежать, если проверять, запущен ли скрипт как программа или импортирован. Это можно сделать с помощью переменной `__name__`, которая определена в любой программе и равна `"__main__"`, если скрипт запущен в качестве главной программы, и имя, если он импортирован. Например, `test.py` может выглядеть вот так:

```
def hello():  
    print('Hello, world!')  
  
def test_function(n):  
    return n ** 2  
  
if __name__ == "__main__":  
    hello()  
    for i in range(10):  
        print(test_function(i))
```

1.4.2. Создание проекта и подключение библиотек в PyCharm

Одним из популярных средств разработки для Python является среда PyCharm. Установить её можно с официального сайта, имеется бесплатная версия.

Рассмотрим процесс создания проекта. Запускаем PyCharm и в окне приветствия выбираем Create New Project.

Далее указываем в поле Location путь расположения создаваемого проекта. Имя конечной директории также является именем проекта. Далее разворачиваем параметры окружения, щёлкая по Project Interpreter. Выбираем New environment using Virtualenv. Путь расположения окружения генерируется автоматически.

После создания проекта можно настроить набор библиотек для выбранного виртуального окружения, которые будем использовать в проекте. С помощью главного меню переходим в настройки File → Settings. Затем Project: project_name → Project Interpreter.

Справа от таблицы имеется панель управления с четырьмя кнопками:

- Кнопка с плюсом добавляет пакет в окружение.
- Кнопка с минусом удаляет пакет из окружения.
- Кнопка с треугольником обновляет пакет.
- Кнопка с глазом включает отображение ранних релизов для пакетов.

Для добавления (установки) библиотеки в окружение нажимаем на плюс. В поле поиска вводим название библиотеки. Дополнительно, через Specify version можно указать версию устанавливаемого пакета и через Options указать параметры.

Для установки библиотек без использования интерфейса PyCharm можно использовать консоль и следующие команды:

Linux:

```
sudo pip3 install <имя модуля>
```

На Windows:

```
pip3 install <имя модуля>
```

Если команда `pip3` не распознана, необходимо указать полный путь до директории, где установлен Python.

1.5. Применение языка программирования Python для анализа и обработки данных

1.5.1. Работа с библиотекой *Pandas*

Pandas – это высокоуровневая Python-библиотека для анализа данных, использующая модуль *NumPy* [4, 5]. Основными структурами в *Pandas* являются *DataFrame* и *Series*.

Series представляет из себя объект, имеющий структуру одномерного массива, но отличительной его чертой является наличие ассоциированных меток, так называемых индексов для каждого элемента из списка. Такая особенность превращает его в ассоциативный массив или словарь в Python.

```
import pandas as pd  
series1 = pd.Series([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

Если индекс явно не задан, то *Pandas* автоматически создаёт *RangeIndex* от 0 до $N - 1$, где N – общее количество элементов. У *Series* есть тип хранимых элементов (например, `int64` для целочисленных значений), а также атрибуты, через которые можно получить список элементов и индексы: *values* и *index* соответственно. Индексы можно задавать явно:

```
series2 = pd.Series([1, 2, 3, 4, 5, 6], index=['a1', 'b2', 'c3', 'd4', 'e5', 'f6'])
```

Сделаем выборку по нескольким индексам и осуществим групповое присваивание:

```
series2[['a1', 'b2', 'd4']] = 10
```

Осуществим фильтрацию *Series* по условию:

```
series2[series2 > 4]
```

У объекта Series и его индекса есть атрибут *name*, задающий имя объекту и индексу соответственно.

```
series3.name = 'series name'
series3.index.name = 'series index'
```

Индекс можно изменить в любое время, присвоив список атрибуту *index* объекта Series:

```
series3.index = ['A', 'B', 'C', 'D']
```

Список с индексами по длине должен совпадать с количеством элементов в Series.

Объект DataFrame является табличной структурой данных. Столбцами в объекте DataFrame выступают объекты Series, строки являются их непосредственными элементами. DataFrame проще всего сконструировать на основе словаря.

Объект DataFrame имеет два индекса: по строкам и по столбцам. Если индекс по строкам явно не задан, то Pandas задаёт целочисленный индекс RangeIndex от 0 до $N - 1$, где N – это количество строк в таблице.

Индекс по строкам можно задать разными способами, например при формировании самого объекта DataFrame:

```
df = pd.DataFrame({
    'position': ['Программисты', 'Бухгалтеры', 'Инженеры', 'Руководители'],
    'salary': [50, 30, 45, 70],
    'staff': [15, 3, 10, 2]
}, index=['CODE', 'ACCO', 'ENG', 'HEAD'])
df.index.name = "Category"
```

Доступ к строкам по индексу возможен несколькими способами:

- *loc* – используется для доступа по строковой метке.
- *iloc* – используется для доступа по числовому значению (начиная от 0).

Можно делать выборку по индексу и интересующим колонкам:

```
df.loc[['CODE', 'HEAD'], 'salary']
```

loc в квадратных скобках может принимать индекс, список индексов, срезы по столбцам и строкам.

Фильтровать DataFrame с помощью булевых массивов, полученных на основе применения условия к значениям целого столбца:

```
df[df.staff > 10][['staff', 'position']]
```

К столбцам можно обращаться, используя атрибут или нотацию словарей Python, т.е. `df.staff` и `df['staff']` – это одно и то же.

Pandas при операциях над DataFrame возвращает новый объект DataFrame, поэтому для сохранения изменений необходимо осуществлять переприсваивание.

Добавление нового столбца:

```
df['sum'] = df['salary'] * df['staff'] * 1000
```

Для удаления столбца используется метод `drop`:

```
df.drop(['sum'], axis='columns')
```

или

```
del df['sum']
```

Переименовывать столбцы нужно через метод `rename`:

```
df = df.rename(columns={'sum': 'summa'})
```

В этом примере перед тем, как переименовать столбец, проверьте, что с него сброшен индекс. Сбросить индексы можно вот так:

```
df.reset_index()
```

Pandas поддерживает все самые популярные форматы хранения данных: csv, excel, sql, буфер обмена, html и многое другое. Чаще всего приходится работать с csv-файлами. Например, чтобы сохранить DataFrame, необходимо написать:

```
df.to_csv('filename.csv')
```

Функции `to_csv` ещё передаются различные аргументы (например, символ разделителя между колонками). Считать данные из csv-файла и превратить в DataFrame можно функцией `read_csv`. Аргумент `sep` указывает разделитель столбцов.

```
df = pd.read_csv('filename.csv', sep=',')
```

С помощью функции *read_sql* Pandas может выполнить SQL-запрос и на основе ответа от базы данных сформировать необходимый DataFrame. В документации библиотеки можно найти инструкции по сохранению и чтению данных в другие форматы.

1.5.2. Обработка отсутствующих данных в pandas

Библиотека NumPy поддерживает использование маскированных массивов, т.е. массивов, к которым присоединены отдельные булевы массивы-маски, предназначенные для маркирования «плохих» или «хороших» данных. Библиотека Pandas могла тоже использовать этот механизм, однако необходимость поддержки и согласования кода привела к тому, что в Pandas было решено использовать для отсутствующих данных индикаторы, а также два уже существующих в Python пустых значения: специальное значение NaN с плавающей точкой и объект None языка Python. У этого решения есть свои недостатки, но на практике в большинстве случаев оно представляет собой удачный компромисс.

Библиотека Pandas рассматривает значения None («ничего») и NaN («бесконечность») как взаимозаменяемые средства указания на отсутствующие или пустые значения. Существует несколько удобных методов для обнаружения, удаления и замены пустых значений в структурах данных библиотеки Pandas, призванных упростить работу с ними [5, 6].

Рассмотрим основные методы по работе с этими типами данных:

- *isnull()* – генерирует булеву маску для отсутствующих значений.
- *notnull()* – противоположность метода *isnull()*.
- *dropna()* – возвращает отфильтрованный вариант данных.
- *fillna()* – возвращает копию данных, в которой пропущенные значения заполнены или восстановлены.

Например, для исключения отсутствующих значений в массиве *data* можно использовать следующую конструкцию:

```
data[data.notnull()]
```

Также существуют следующие методы: *dropna()* (отбрасывающий отсутствующие значения) и *fillna()* (заполняющий значения).

По умолчанию *dropna()* отбрасывает все строки, в которых присутствует хотя бы одно пустое значение.

Иногда предпочтительнее вместо отбрасывания пустых значений заполнить их каким-то допустимым значением. Это значение может быть фиксированным, например нулем, или полученным на основе каких-либо вычислений.

Это можно сделать путём замены в исходных данных, используя результат метода `isnull()` в качестве маски. Однако для упрощения библиотека Pandas предоставляет метод `fillna()`, возвращающий копию массива с замененными пустыми значениями.

Можно заполнить отсутствующие элементы одним фиксированным значением, например нулями: `data.fillna(0)`

Можно задать параметр заполнения по направлению «вперёд», копируя предыдущее значение в следующую ячейку `data.fillna(method='ffill')` или по направлению «назад» – `data.fillna(method='bfill')`.

1.6. Визуализация данных на Python с помощью библиотеки Matplotlib

1.6.1. Основы библиотеки Matplotlib

Библиотека Matplotlib – это библиотека двумерной графики для языка программирования Python, с помощью которой можно создавать высококачественные рисунки различных форматов [7].

Главной единицей (объектом самого высокого уровня) при работе с Matplotlib является рисунок (класс Figure). Любой рисунок имеет следующую вложенную структуру:

Figure (Рисунок) включает Axes (Область рисования), которые включают Axis (Координатная ось).

Рисунок является объектом самого верхнего уровня, на котором располагаются одна или несколько областей рисования (Axes), элементы рисунка Artists (заголовки, легенда и т.д.) и основа-холст (Canvas). На рисунке может быть несколько областей рисования Axes, но данная область рисования Axes может принадлежать только одному рисунку Figure.

Область рисования является объектом среднего уровня, это часть изображения с пространством данных. Каждая область рисования Axes содержит две (или три в случае трёхмерных данных) координатные оси (Axis объектов), которые упорядочивают отображение данных.

Координатная ось тоже является объектом среднего уровня, которая определяет область изменения данных, на неё наносятся деления `ticks` и подписи к делениям `ticklabels`. Расположение делений определяется объектом `Locator`, а подписи делений обрабатывает объект `Formatter`.

Элементы рисунка `Artists` включают практически всё, что отображается на рисунке, даже объекты `Figure`, `Axes` и `Axis`. Элементы рисунка `Artists` включают в себя такие простые объекты, как текст (`Text`), плоская линия (`Line2D`), фигура (`Patch`) и другие.

Когда происходит отображение рисунка, все элементы рисунка `Artists` наносятся на основу-холст (`Canvas`). Большая часть из них связывается с областью рисования `Axes`. Также элемент рисунка не может совместно использоваться несколькими областями `Axes` или быть перемещён с одной на другую.

Для работы с графиками мы будем использовать интерфейс `pyplot`, включающий множество готовых решений. Существует неофициальный стандарт вызова `pyplot` в Python, который можно увидеть во многих примерах:

```
import matplotlib.pyplot as plt
```

В `Matplotlib` работает правило «текущей области» ("current axes"), которое означает, что все графические элементы наносятся на текущую область рисования. Несмотря на то, что областей рисования может быть несколько, одна из них всегда является текущей.

Создание графики нужно начинать именно с создания рисунка. Создать рисунок в `Matplotlib` означает задать форму, размеры и свойства основы-холста (`Canvas`), на котором будет создаваться будущий график.

Создать рисунок позволяет метод `plt.figure()`. После вызова любой графической команды, т.е. функции, которая создаёт какой-либо графический объект, например, `plt.scatter()` или `plt.plot()`, всегда существует хотя бы одна область для рисования (по умолчанию прямоугольной формы). Рассмотрим пример.

```
import matplotlib.pyplot as plt
```

```
fig = plt.figure()
ax = fig.add_axes([0, 0, 1, 1])
print(type(ax))
plt.scatter(1.0, 1.0)
plt.savefig('example 1.png', fmt='png')
```



```

fig = plt.figure()
ax = fig.add_axes([0, 0, 1, 1], polar=True)
plt.scatter(0.0, 0.5)
plt.savefig('example 2.png', fmt='png')

plt.show()

```

Чтобы результат рисования, т.е. текущее состояние рисунка, отразилось на экране, можно воспользоваться командой *plt.show()*. Будут показаны все рисунки, которые были созданы.

Чтобы сохранить получившийся рисунок нужно воспользоваться методом *plt.savefig()*. Он сохраняет текущую конфигурацию текущего рисунка в графический файл с некоторым расширением (png, jpeg, pdf и др.), который можно задать через параметр *fmt*. Поэтому её нужно вызывать в конце исходного кода, после вызова всех других команд. Если в python-скрипте создать несколько рисунков *figure* и попытаться сохранить их одной командой *plt.savefig()*, то будет сохранён последний рисунок *figure*.

Часто на рисунок наносятся линии вспомогательной сетки (grid). В *pyplot* она вызывается командой *plt.grid()*. Вспомогательная сетка связана с делениями координатных осей (ticks), которые определяются автоматически. В *matplotlib* существуют главные деления (major ticks) и вспомогательные (minor ticks) для каждой координатной оси.

Рассмотрим более сложный пример, включающий рисование графика, разметку координатных осей, создание надписей в произвольных местах рисунка.

```

import matplotlib.pyplot as plt
import numpy as np

i = 0.1
x = np.arange(-5, 5, i)
y = x**3
fig = plt.figure()
plt.plot(x, y)
plt.text(np.pi-2.5, 50, 'Надпись 1', fontsize=16, bbox=dict(edgecolor='w',
color='w'))
plt.text(0.1, 0, 'Надпись на оси Y', fontsize=8, bbox=dict(edgecolor='w',
color='w'), rotation=90)

```

```
plt.text(-3, 1, 'Надпись на оси X', fontsize=8, bbox=dict(edgecolor='w',
color='w'))
plt.title("ЗАГОЛОВОК")
plt.ylabel('Подпись оси Y')
plt.xlabel('Подпись оси X')
plt.text(-4, -64, 'отметка на графике', fontsize=8, bbox=dict(edgecolor='w',
color='w'), rotation=90)
plt.grid(True)
plt.show()
```

Результат работы представлен на рис. 1.1.



Рис. 1.1. Рисование графика с помощью Matplotlib

Большинство объектов из библиотеки Matplotlib обладают набором общих атрибутов, изменяя которые, можно регулировать оформление, размеры или иные свойства данных объектов.

Наиболее часто встречаемые названия параметров изменения свойств графических объектов перечислены ниже:

- color/colors/c – цвет;

- `linewidth/linewidths` – толщина линии;
- `linestyle` – тип линии;
- `alpha` – степень прозрачности (от полностью прозрачного 0 до непрозрачного 1);
- `fontsize` – размер шрифта;
- `marker` – тип маркера;
- `s` – размер маркера в методе `plt.scatter` (только цифры);
- `rotation` – поворот строки на X градусов.

1.6.2. Диаграммы и объединённые графики

Библиотека `Matplotlib` предоставляет широкие возможности по созданию различных фигур, графиков и диаграмм, что упрощает визуализацию собранных данных и построение научной графики [5, 7]. Ниже перечислены наиболее распространённые команды (подробное описание для каждой из них можно найти в официальной документации):

- `plt.plot()` – ломаная линия;
- `plt.scatter()` – маркер или точечное рисование;
- `plt.text()` – нанесение текста;
- `plt.bar()`, `plt.barh()`, `plt.barbs()`, `broken_barh()` – столбчатая диаграмма;
- `plt.hist()`, `plt.hist2d()`, `plt.hlines` – гистограмма;
- `plt.pie()` – круговая диаграмма;
- `plt.boxplot()` – диаграмма размаха;
- `plt.errorbar()` – оценка погрешности;
- `plt.contour()` – изолинии;
- `plt.contourf()` – изолинии с послойной окраской;
- `plt.pcolor()`, `plt.pcolormesh()` – псевдоцветное изображение матрицы;
- `plt.imshow()` – вставка графики (пиксели + сглаживание);
- `plt.matshow()` – отображение данных в виде квадратов;
- `plt.fill()` – заливка многоугольника;
- `plt.fill_between()`, `plt.fill_betweenx()` – заливка между двумя линиями;

Рассмотрим пример работы с диаграммами различного типа.

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

s = ['one', 'two', 'three ', 'four', 'five']
x = [1, 2, 3, 4, 5]
z = np.random.random(100)
z1 = [12, 11, 8, 14, 21]
z2 = [3, 9, 14, 7, 4]

fig = plt.figure()
plt.bar(x, z1)
plt.title('bar chart')
plt.grid(True)

fig = plt.figure()
plt.hist(z)
plt.title('histogramm')
plt.grid(True)

fig = plt.figure()
plt.pie(x, labels=s)
plt.title('pie chart')

fig = plt.figure()
plt.boxplot([z1, z2])
plt.title('box chart')
plt.grid(True)

fig = plt.figure()
plt.errorbar(x, z1, xerr=1, yerr=0.5)
plt.title('error bar chart')
plt.grid(True)

plt.show()
```

Каждая диаграмма из примера формируется на отдельном рисунке. Не всегда это бывает удобно. В Matplotlib реализовано несколько способов создания рисунков с несколькими областями для диаграмм:

- *fig.add_axes()* – базовый метод, удобен при создании диаграммы-врезки;
- *fig.add_subplot()* – добавление одного *subplot* на рисунок. Удобно для отображения 2–3 диаграмм;
- *plt.subplot()* – аналогичный предыдущему по результату метод для *pyplot*;
- *plt.subplots()* – удобный метод для автоматизированного создания нескольких *subplots*;
- *plt.GridSpec()* – метод для объединения ячеек *subplots* в более сложные конфигурации. Позволяет создавать разные по форме *subplots* на рисунке.

Первые два метода (с приставкой *add_*) являются более низкоуровневыми и для их вызова требуется объект *Figure*. Последние три метода реализованы в *pyplot*-интерфейсе. Каждый из этих методов создаёт один или более экземпляров типа *subplot* или *axes*.

Для создания множества областей удобно не просто добавлять их на рисунок последовательно, по одному, а разбить рисунок на несколько областей рисования. Это позволяет сделать метод *plt.subplots()*, в котором указывается число строк и столбцов создаваемой таблицы. Каждая ячейка этой таблицы является экземпляром *subplots*. Метод возвращает объект типа *figure* и массив из созданных *subplots*.

В примере ниже представлено размещение рассмотренных выше диаграмм на одном изображении в сетке 2×3.

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

s = ['one', 'two', 'three ', 'four', 'five']
x = [1, 2, 3, 4, 5]
z = np.random.random(100)
z1 = [12, 11, 8, 14, 21]
z2 = [3, 9, 14, 7, 4]

fig, subplots = plt.subplots(nrows=2, ncols=3, sharex=True, sharey=True)

fig = plt.subplot(2, 3, 1)
plt.bar(x, z1)
plt.title('bar chart')
plt.grid(True)
```

```

fig = plt.subplot(2, 3, 2)
plt.hist(z)
plt.title('histogramm')
plt.grid(True)

fig = plt.subplot(2, 3, 3)
plt.pie(x, labels=s)
plt.title('pie chart')

fig = plt.subplot(2, 3, 4)
plt.boxplot([z1, z2])
plt.title('box chart')
plt.grid(True)

fig = plt.subplot(2, 3, 5)
plt.errorbar(x, z1, xerr=1, yerr=0.5)
plt.title('error bar chart')
plt.grid(True)

fig = plt.subplot(2, 3, 6)
plt.plot([1, 2, 3, 4, 5], [4, 5, 7, 8, 9])

plt.tight_layout(h_pad=-0.15, w_pad=-0.2)

plt.show()

```

Полученный результат представлен на рис. 1.2. Необходимо отметить, что при размещении множества графиков на одной фигуре может пострадать их читаемость.

Библиотека `matplotlib` содержит ещё достаточно много функций и методов, позволяющих спроектировать визуализацию данных нужным вам способом. Подробнее об всех возможностях библиотеки вы можете узнать в официальной документации:

<https://matplotlib.org/>

Однако даже тех возможностей, что мы рассмотрели, достаточно для визуализации графиков функций, диаграмм, анализа и представления данных.

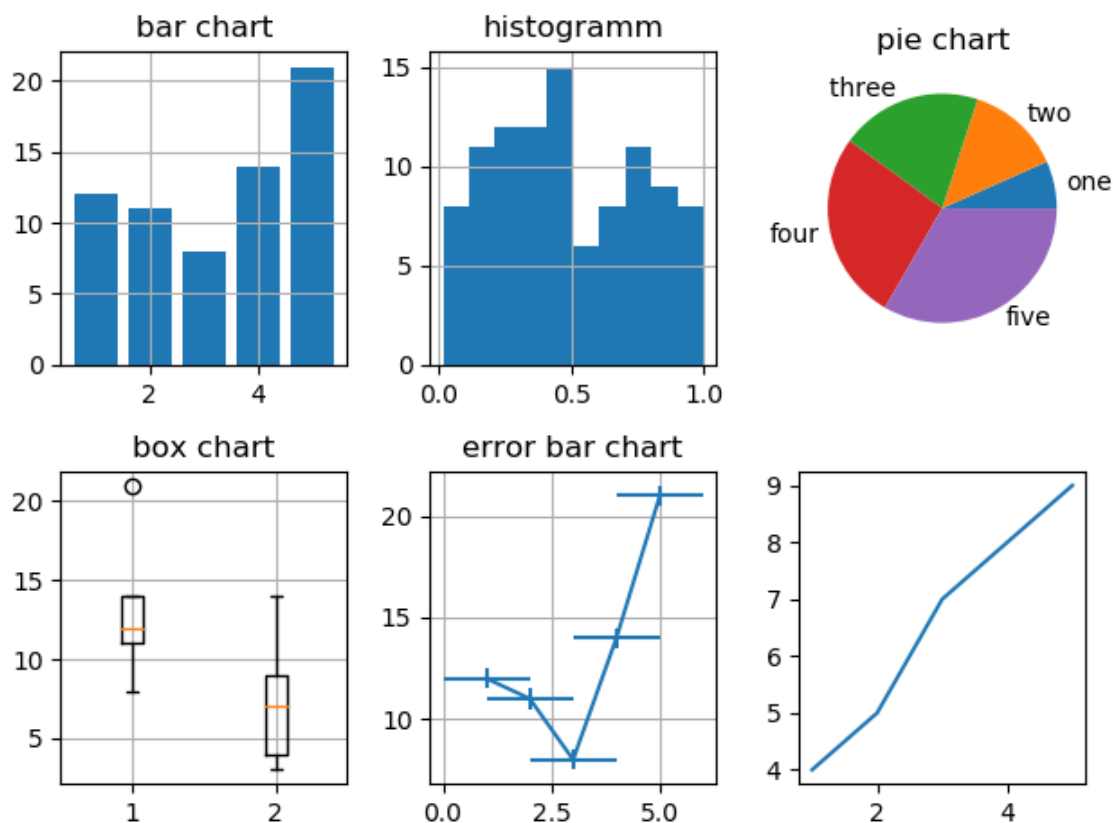


Рис. 1.2. Размещение нескольких графиков с помощью Matplotlib

Вопросы для закрепления

1. Дайте определение терминам «информация», «информационная система», «офисные технологии», «облачные технологии».
2. Назовите основные типы данных в Python.
3. Приведите примеры основных операторов в Python.
4. Приведите пример создания функций в Python.
5. Дайте определение понятиям наследование, полиморфизм, инкапсуляция и приведите примеры.
6. Приведите примеры используемых Вами библиотек Python.
7. Опишите основные возможности библиотеки Pandas.
8. Опишите основные возможности библиотеки Matplotlib.

2. ОБРАБОТКА ИНФОРМАЦИИ В ОФИСНЫХ СИСТЕМАХ

Важной задачей при реализации делопроизводства и электронного документооборота является автоматизация процессов работы с документами. В рамках данного раздела будут рассмотрены основные подходы к обработке различных типов документов с помощью Python [8].

2.1. Обработка офисных документов DOCX на Python

Работа с документами Microsoft Word (формат .docx) осуществляется с помощью библиотеки python-docx. Данный модуль позволяет создавать и изменять документы MS Word с расширением .docx.

Обратите внимание, что при установке модуля надо вводить python-docx, а не docx. В то же время при импортировании модуля python-docx следует использовать `import docx`, а не `import python-docx`.

Рассмотрим процесс чтения документов. Файлы с расширением .docx в python-docx обладают иерархической внутренней структурой. На самом верхнем уровне объект Document представляет собой весь документ. Объект Document содержит список объектов Paragraph, которые представляют собой абзацы документа. Каждый из абзацев содержит список, состоящий из одного или нескольких объектов Run, представляющих собой фрагменты текста с различными стилями форматирования.

```
import docx
print(len(doc.paragraphs))
print(doc.paragraphs[0].text)
print(doc.paragraphs[1].text)
print(doc.paragraphs[1].runs[0].text)
text = []
for paragraph in doc.paragraphs:
    text.append(paragraph.text)
print('\n'.join(text))
```

В документах MS Word применяются два типа стилей: стили абзацев, которые могут применяться к объектам Paragraph, стили символов, которые могут

применяться к объектам Run. Как объектам Paragraph, так и объектам Run можно назначать стили, присваивая их атрибутам style-значение в виде строки. Этой строкой должно быть имя стиля. Если для стиля задано значение None, то у объекта Paragraph или Run не будет связанного с ним стиля.

Примеры стилей абзацев: Normal, Body Text, Heading, Caption, List Number.

Примеры стилей символов: Strong, Book, Title, Default.

Отдельные фрагменты текста, представленные объектами Run, могут подвергаться дополнительному форматированию с помощью атрибутов. Для каждого из этих атрибутов может быть задано одно из трех значений: True (атрибут активизирован), False (атрибут отключен) и None (применяется стиль, установленный для данного объекта Run).

Список атрибутов: bold – Полужирное начертание; underline – Подчёркнутый текст; italic – Курсивное начертание; strike – Зачёркнутый текст.

Для сохранения документа используется команда *save*:

```
doc.save('new.docx')
```

Рассмотрим пример, в котором откроем документ и перенесём его стили в другой файл:

```
import docx  
doc1 = docx.Document('example.docx')  
doc2 = docx.Document('restyled.docx')  
styles = []  
for paragraph in doc1.paragraphs:  
    styles.append(paragraph.style)  
for i in range(len(doc2.paragraphs)):  
    doc2.paragraphs[i].style = styles[i]  
doc2.save('doc2.docx')
```

Рассмотрим создание документов. Добавление абзацев осуществляется вызовом метода *add_paragraph()* объекта Document. Для добавления текста в конец существующего абзаца надо вызвать метод *add_run()* объекта Paragraph:

```
import docx  
doc = docx.Document()  
doc.add_paragraph('Тест')  
par1 = doc.add_paragraph('2 тест.')
```

```

par2 = doc.add_paragraph('3 месм.')
par1.add_run(' добавлено во второй абзац.')
par2.add_run(' добавлено во второй абзац жирным шрифтом.').bold = True
doc.save('test.docx')

```

Оба метода, `add_paragraph()` и `add_run()`, принимают необязательный второй аргумент, содержащий строку стиля, например:

```

doc.add_paragraph('заголовок', 'Title')

```

Вызов метода `add_heading()` приводит к добавлению абзаца, отформатированного в соответствии с одним из возможных стилей заголовков. Аргументами метода `add_heading()` являются строка текста и целое число от 0 до 4. Например:

```

doc.add_heading('Заголовок', 0)

```

Чтобы добавить разрыв строки (а не добавлять новый абзац), нужно вызвать метод `add_break()` объекта `Run`. Если же требуется добавить разрыв страницы, то методу `add_break()` надо передать значение `docx.enum.text.WD_BREAK.PAGE` в качестве единственного аргумента. Пример:

```

import docx
doc = docx.Document()
doc.add_paragraph('1 страница')
doc.paragraphs[0].runs[0].add_break(docx.enum.text.WD_BREAK.PAGE)
doc.add_paragraph('2 страница')
doc.save('pages.docx')

```

Метод `add_picture()` объекта `Document` позволяет добавлять изображения в конце документа. Например, добавим в конец документа изображение `test.jpg` шириной 5 сантиметров:

```

import docx
doc = docx.Document()
doc.add_paragraph('Это первый абзац')
doc.add_picture('test.jpg', width = docx.shared.Cm(5))
doc.save('test.docx')

```

Именованные аргументы `width` и `height` задают ширину и высоту изображения. Если их не указывать, то значения этих аргументов будут определяться размерами самого изображения.

Для добавления таблицы в документ используются методы `add_table`, создающий объект таблицы с заданным количеством строк и столбцов. Заполнение таблицы осуществляется через метод `cell(row, col)` для элемента строки `row` и столбца `col`. Пример:

```
import docx
doc = docx.Document()
table = doc.add_table(rows = 5, cols = 4)
table.style = 'Table Grid'
for row in range(5):
    for col in range(4):
        cell = table.cell(row, col)
        cell.text = str(row + 1) + str(col + 1)
doc.save('table.docx')
```

Рассмотрим также пример чтения данных из таблицы:

```
import docx

doc = docx.Document('table.docx')
table = doc.tables[0]
for row in table.rows:
    string = ""
    for cell in row.cells:
        string = string + cell.text + ' '
    print(string)
```

Таким образом, рассмотренные методы и примеры позволяют автоматизировать создание, чтение, изменение документов формата `.docx`.

2.2. Обработка таблиц XLSX на Python

Рассмотрим работу с электронными таблицами Excel в формате `.xlsx`. Структура документов данного формата представлена книгами, разделёнными на некоторое количество листов [9]. Открытый и отображаемый на экране лист считается активным. Лист состоит из столбцов (адресуемых с помощью букв, начиная с A) и строк (адресуемых с помощью цифр, начиная с 1).

В Python для работы с электронными таблицами используется несколько библиотек. Рассмотрим некоторые из них.

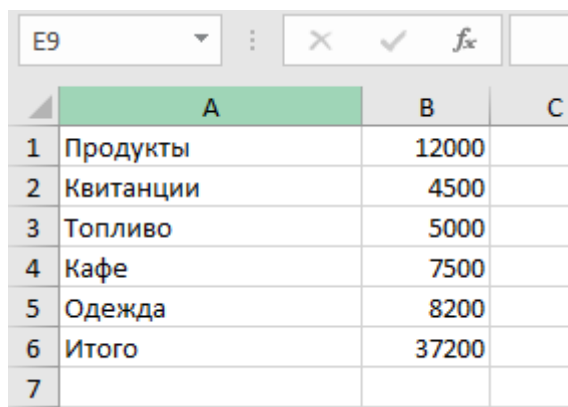
2.2.1. Библиотека *xlsxwriter*

Рассмотрим два простых примера. В первом создадим простейшую таблицу:

```
import xlsxwriter
workbook = xlsxwriter.Workbook('Расходы.xlsx')
worksheet = workbook.add_worksheet()
data = [('Продукты', 12000), ('Квитанции', 4500), ('Топливо', 5000), ('Кафе', 7500), ('Одежда', 8200)]

for row, (i, j) in enumerate(data):
    worksheet.write(row, 0, i)
    worksheet.write(row, 1, j)
row += 1
worksheet.write(row, 0, 'Итого')
worksheet.write(row, 1, '=SUM(B1:B5)')
workbook.close()
```

Результат работы данного примера представлен на рис. 2.1.



	A	B	C
1	Продукты	12000	
2	Квитанции	4500	
3	Топливо	5000	
4	Кафе	7500	
5	Одежда	8200	
6	Итого	37200	
7			

Рис. 2.1. Создание простейшей таблицы в Excel на Python

Для создания листа с определённым именем необходимо указать его при вызове метода *add_worksheet*:

```
worksheet = workbook.add_worksheet("Новый лист")
```

Во втором примере построим простейшую диаграмму:

```
import xlswriter
```

```
workbook = xlswriter.Workbook('Расходы.xlsx')
```

```
worksheet = workbook.add_worksheet()
```

```
data = [('Продукты', 12000), ('Квитанции', 4500), ('Топливо', 5000), ('Кафе', 7500), ('Одежда', 8200)]
```

```
for row, (i, j) in enumerate(data):
```

```
    worksheet.write(row, 0, i)
```

```
    worksheet.write(row, 1, j)
```

```
chart = workbook.add_chart({'type': 'pie'})
```

```
chart.add_series({'values': '=Sheet1!$B$1:$B$5'})
```

```
worksheet.insert_chart('C1', chart)
```

```
workbook.close()
```

Результат выполнения данного скрипта представлен на рис. 2.2.

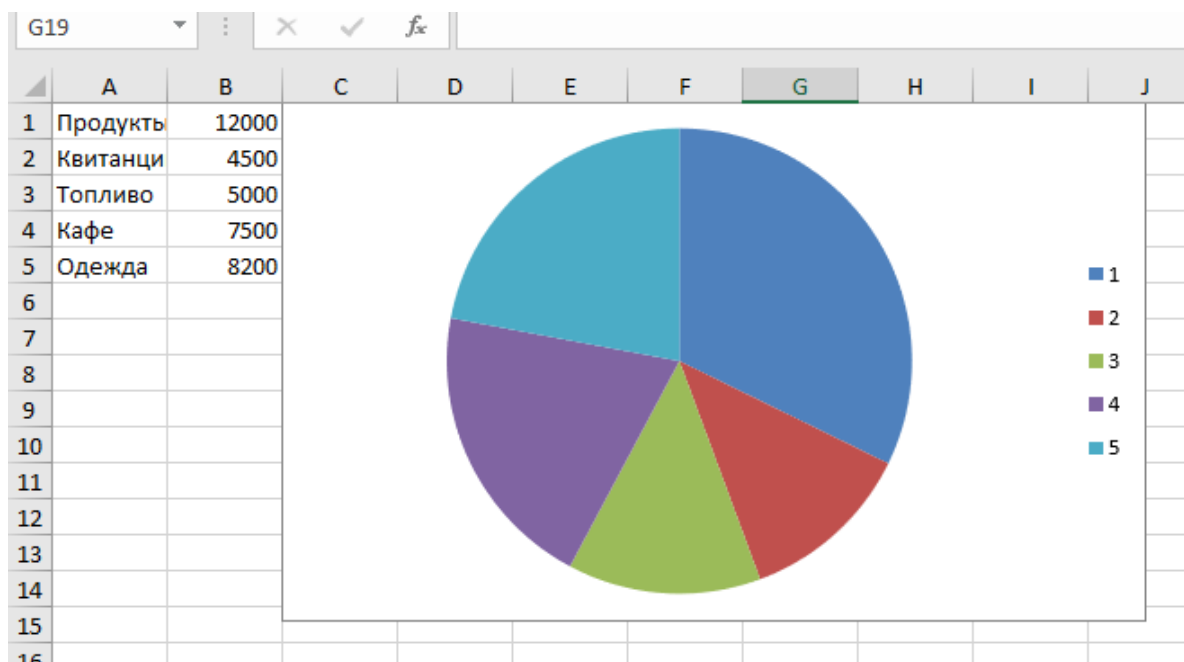


Рис. 2.2. Создание диаграммы в xlswriter

Однако, модуль xlswriter не поддерживает редактирование и чтение документов. Поэтому рассмотрим ещё одну библиотеку для работы с электронными таблицами.

2.2.2. Библиотека OpenPyXL

Модуль OpenPyXL обеспечивает широкие возможности по чтению и редактированию документов. Рассмотрим его основные возможности.

Откроем созданную ранее таблицу для чтения:

```
import openpyxl
workbook = openpyxl.load_workbook('Расходы.xlsx')
sheet = workbook.active
sheets = workbook.sheetnames
for sheet in sheets:
    print(sheet)
sheet = workbook.active
print(sheet['A1'].value)
print(sheet['B1'].value)
```

Таким образом, мы прочитаем список всех листов, выберем из них активный и выведем значения двух ячеек с него. Для того чтобы выбрать другой лист и вывести значение ячейки с него, используем следующие команды:

```
sheet2 = workbook ['Имя листа']
print(sheet2['A1'].value)
```

Для изменения активного листа используем:

```
workbook.active = <номер листа, начиная с 0>
```

Чтобы переименовать лист:

```
sheet.title = 'Новое имя'
```

Рассмотрим работу с ячейками. Объект Cell имеет атрибут *value*, который содержит значение, хранящееся в ячейке. Объект Cell также содержит атрибуты *row*, *column* и *coordinate*, которые предоставляют информацию о расположении данной ячейки в таблице.

```
cell = sheet['B1']
print('Строка: ' + str(cell.row))
print('Столбец: ' + cell.column)
print('Ячейка: ' + cell.coordinate)
print('Значение: ' + cell.value)
```

К отдельной ячейке можно также обращаться с помощью метода *cell()* объекта *Worksheet*, передавая ему именованные аргументы *row* и *column*. Первому столбцу или первой строке соответствует число 1, а не 0:

```
cell = sheet.cell(row= 2, column= 2)
```

Размер листа можно получить с помощью атрибутов *max_row* и *max_column* объекта *Worksheet*:

```
rows = sheet.max_row
cols = sheet.max_column
for i in range(1, rows + 1):
    string = ''
    for j in range(1, cols + 1):
        cell = sheet.cell(row = i, column = j)
        string = string + str(cell.value) + ' '
```

Чтобы преобразовать буквенное обозначение столбца в цифровое, следует вызвать функцию *openpyxl.utils.column_index_from_string()*. Для обратного преобразования используется функция *openpyxl.utils.get_column_letter()*.

Для вызова этих функций загружать рабочую книгу необязательно:

```
get_column_letter(1) # 'A'
column_index_from_string('A') # 1
```

Используя срезы объектов *Worksheet*, можно получить все объекты *Cell*, принадлежащие определённой строке, столбцу или прямоугольной области.

```
sheet['A1':'B5']
```

Выводим значения второй колонки:

```
sheet['B']
```

Выводим строки с первой по пятую:

```
sheet[1:5]
```

Для доступа к ячейкам конкретной строки или столбца также можно воспользоваться атрибутами *rows* и *columns* объекта *Worksheet*.

```
list(sheet.rows)
for row in sheet.rows:
    print(row)
```

Выводим значения всех ячеек листа:

```
for row in sheet.rows:
    string = ''
    for cell in row:
        string = string + str(cell.value) + ' '
    print(string)
```

Далее рассмотрим запись файлов Excel с помощью данной библиотеки.

Метод `create_sheet()` возвращает новый объект `Worksheet`, который по умолчанию становится последним листом книги. С помощью именованных аргументов `title` и `index` можно задать имя и индекс нового листа.

Метод `remove()` принимает в качестве аргумента не строку с именем листа, а объект `Worksheet`. Если известно только имя листа, который надо удалить, используйте `workbook[sheetname]`. Ещё один способ удалить лист – использовать инструкцию `del workbook[sheetname]`.

Чтобы сохранить изменения после добавления или удаления листа рабочей книги, необходимо вызвать метод `save()`.

Пример:

```
import openpyxl

workbook = openpyxl.Workbook()
workbook.create_sheet(title='Первый лист', index=0)
workbook.remove(workbook['Первый лист'])
workbook.create_sheet(title='Лист 1', index=0)
sheet = workbook['Лист 1']
sheet['A1'] = 'Тест'
for row in range(4, 8):
    for col in range(2, 7):
        value = str(row) + str(col)
        cell = sheet.cell(row = row, column = col)
        cell.value = value

workbook.save('example.xlsx')
```

Результат данного скрипта представлен на рис. 2.3.

	A	B	C	D	E	F
1	Тест					
2						
3						
4		42	43	44	45	46
5		52	53	54	55	56
6		62	63	64	65	66
7		72	73	74	75	76
8						

Рис. 2.3. Заполнение таблицы в OpenPyXL

Можно также добавлять строки целиком:

```
sheet.append([1, 2, 3])
sheet.append(['Четвертый', 'Пятый', 'Шестой'])
sheet.append(['мест 7', 'мест 8', 'мест 9'])
```

Для настройки шрифтов, используемых в ячейках, необходимо импортировать функцию *Font()* из модуля *openpyxl.styles*:

```
from openpyxl.styles import Font
```

Ниже приведён пример создания новой рабочей книги, в которой для шрифта, используемого в ячейке A1, устанавливается шрифт Arial, фиолетовый цвет, курсивное полужирное начертание и размер 15 пункта. Если стилевое оформление применяется к большому количеству ячеек используют именованные стили. Рассмотрим их использование на примере ячейки A2.

```
import openpyxl
from openpyxl.styles import NamedStyle, Font, Border, Side

new_style = NamedStyle(name='test_style')
new_style.font = Font(bold=True, size=20)
border = Side(style='thick', color='000000')
new_style.border = Border(left=border, top=border, right=border, bottom=border)

workbook = openpyxl.Workbook()
workbook.create_sheet(title='Первый лист', index=0)
workbook.remove(workbook['Первый лист'])
```

```

workbook.create_sheet(title='Лист 1', index=0)
sheet = workbook['Лист 1']
workbook.add_named_style(new_style)

font = Font(name='Arial', size=15, bold=True, italic=True, color='8F10AA')
sheet['A1'].font = font
sheet['A1'] = 'Test1'

sheet['A2'] = 'Test2'
sheet['A2'].style = 'test_style'

workbook.save('example.xlsx')

```

Итоговый стиль ячеек представлен на рис. 2.4.

	A	B	C
1	Test1		
2	Test2		
3			

Рис. 2.4. Работа со стилями в OpenPyXL

При работе с таблицами Excel одной из важнейших задач является применение формул для решения экономических, математических, статистических или иных задач. Формулы, начинающиеся со знака равенства, позволяют устанавливать для ячеек значения, рассчитанные на основе значений в других ячейках. Например, следующая запись сохранит `=SUM(A1:A8)` в качестве значения в ячейке B1, т.е. сумму элементов из столбца A:

```
sheet['B1'] = '=SUM(A1:A8)'
```

Формула Excel – это математическое выражение, которое создаётся для вычисления результата и которое может зависеть от содержимого других ячеек. Формула в ячейке Excel может содержать данные, ссылки на другие ячейки, а также обозначение действий, которые необходимо выполнить.

Использование ссылок на ячейки позволяет пересчитывать результат по формулам, когда происходят изменения содержимого ячеек, включённых в формулы. Формулы Excel начинаются со знака равно, а скобки () могут использоваться для определения порядка математических операций.

Хранящуюся в ячейке формулу можно читать, как любое другое значение. Однако, если нужно получить результат расчёта по формуле, а не саму формулу, то при вызове функции *load_workbook()* ей следует передать именованный аргумент *data_only* со значением *True*.

С помощью модуля *OpenPyXL* можно задавать высоту строк и ширину столбцов таблицы, закреплять их на месте (чтобы они всегда были видны на экране), полностью скрывать из виду, объединять ячейки.

Рассмотрим настройку высоты строк и ширины столбцов. Объекты *Worksheet* имеют атрибуты *row_dimensions* и *column_dimensions*, которые управляют высотой строк и шириной столбцов.

```
sheet['A1'] = 'Высокая строка'
sheet['B2'] = 'Широкий столбец'
sheet.row_dimensions[1].height = 100
sheet.column_dimensions['B'].width = 50
```

Для указания высоты строки разрешено использовать целые или вещественные числа в диапазоне от 0 до 409. Для указания ширины столбца можно использовать целые или вещественные числа в диапазоне от 0 до 255. Столбцы с нулевой шириной и строки с нулевой высотой невидимы для пользователя.

Ячейки, занимающие прямоугольную область, могут быть объединены в одну ячейку с помощью метода *merge_cells()* рабочего листа:

```
sheet.merge_cells('A1:B3')
sheet.merge_cells('C7:F7')
```

Чтобы отменить слияние ячеек, надо вызвать метод *unmerge_cells()*:

```
sheet.unmerge_cells('A1:B3')
```

Если размер таблицы настолько велик, что её нельзя увидеть целиком, можно заблокировать несколько верхних строк или крайних слева столбцов в их позициях на экране. В этом случае пользователь всегда будет видеть заблокированные заголовки столбцов или строк, даже если он прокручивает таблицу на экране.

У объекта `Worksheet` имеется атрибут `freeze_panes`, значением которого может служить объект `Cell` или строка с координатами ячеек. Все строки и столбцы, расположенные выше и левее, будут заблокированы.

```
sheet.freeze_panes = 'C1' # заблокирует столбцы A и B
sheet.freeze_panes = None # убрать заблокированные области
```

Модуль `OpenPyXL` поддерживает создание гистограмм, графиков, а также точечных и круговых диаграмм с использованием данных, хранящихся в электронной таблице. Чтобы создать диаграмму, необходимо выполнить следующие действия:

- создать объект `Reference` на основе ячеек в пределах выделенной прямоугольной области;
- создать объект `Series`, передав функции `Series()` объект `Reference`;
- создать объект `Chart`;
- дополнительно можно установить значения переменных `drawing.top`, `drawing.left`, `drawing.width`, `drawing.height` объекта `Chart`, определяющих положение и размеры диаграммы;
- добавить объект `Chart` в объект `Worksheet`.

Объекты `Reference` создаются путём вызова функции `openpyxl.charts.Reference()`, принимающей пять аргументов: объект `Worksheet`, содержащий данные диаграммы; два целых числа (строка, столбец), представляющих верхнюю левую ячейку выделенной прямоугольной области, в которых содержатся данные диаграммы; два целых числа (строка, столбец), представляющих нижнюю правую ячейку. Пример:

```
from openpyxl import Workbook
from openpyxl.chart import BarChart, Reference

workbook = Workbook()
workbook.create_sheet(title='1 лист', index=0)
sheet = workbook['1 лист']
sheet['A1'] = 'Data'
for i in range(1, 10):
    cell = sheet.cell(row=i + 1, column=1)
    cell.value = i * 5 - i ** 2
```

```

chart = BarChart()
chart.title = 'Bar chart'
data = Reference(sheet, min_col=1, min_row=1, max_col=1, max_row=10)
chart.add_data(data, titles_from_data=True)
sheet.add_chart(chart, 'C2')
workbook.save('example.xlsx')

```

Результат скрипта представлен на рис. 2.5.

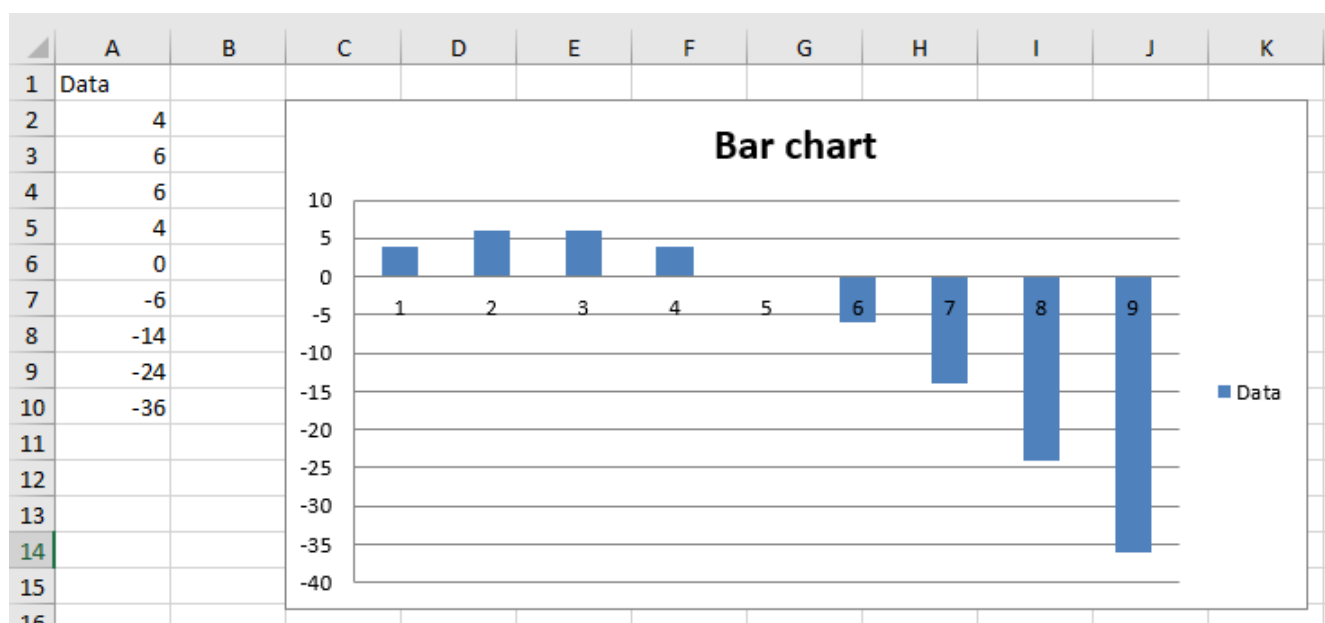


Рис. 2.5. Создание диаграммы в OpenPyXL

Аналогично можно создавать графики, точечные и круговые диаграммы, вызывая методы:

```

openpyxl.chart.LineChart()
openpyxl.chart.ScatterChart()
openpyxl.chart.PieChart()

```

2.3. Работа с PDF на Python

PDF является одним из популярных форматов офисных документов. Для работы с ним на Python также существует несколько библиотек. Рассмотрим библиотеку PyPDF2 (существует её более новая версия PyPDF4).

Мы используем класс PdfFileReader, чтобы открыть и прочитать документ (метод *getDocumentInfo*), извлечь количество страниц (метод *getNumPages*) и сам текст (*extractText*). PyPDF2/4 начинает считать страницы с 0, и поэтому вызов *pdf.getPage(0)* извлекает первую страницу документа. Извлечённый текст печатается непрерывно. Здесь нет ни абзацев, ни разделений предложений.

```
from PyPDF4 import PdfFileReader

pdf_document = "example.pdf"
with open(pdf_document, "rb") as filehandle:
    pdf = PdfFileReader(filehandle)
    info = pdf.getDocumentInfo()
    pages = pdf.getNumPages()
    print(info)
    print("количество страниц:", pages)
    page_1 = pdf.getPage(0)
    print(page_1)
    print(page_1.extractText())
```

Недостатком PyPDF2/4 является невозможность извлечь кириллицу из документа. Поэтому рассмотрим второй вариант чтения документа с помощью PyMuPDF. Отображение информации о документе, печать количества страниц и извлечение текста из документа PDF выполняется аналогично PyPDF2/4. Импортимый модуль имеет имя *fitz*.

```
import fitz

pdf_doc = "example.pdf"
doc = fitz.open(pdf_doc)
print("количество страниц: %i" % doc.pageCount)
print(doc.metadata)
page1 = doc.loadPage(0)
text1 = page1.getText("text")
print(text1)
```

Метод *fitz.open()* используется как для открытия существующих документов (тогда ему нужно передать его имя), так и для создания новых (если аргументы отсутствуют).

Особенностью PyMuPDF является то, что он сохраняет исходную структуру документа без изменений – целые абзацы с разрывами строк сохраняются такими же, как в PDF-документе.

Можно перебрать все страницы в документе с помощью различных циклов:

```
for page in doc: # прямой порядок  
for page in reversed(doc): # обратный порядок  
for page in doc.pages(start, stop, step): # в интервале с шагом
```

Можно загрузить список ссылок (оглавление) документа:

```
links = page.getLinks()
```

Рассмотрим основные команды для редактирования структуры документов.

Добавление пустой страницы на заданный номер с выбранными размерами страницы:

```
newPage(pno=<номер страницы>, width=<ширина>, height=<высота>)
```

Добавление страницы с текстом, заданным размером и типом шрифта, размером страницы:

```
insertPage(pno, text="test", fontsize=11, width=595, height=842, font-  
name="helv", fontfile=None, color=None)
```

Удалить страницу (при аргументе=-1 – удалить последнюю страницу):

```
deletePage(pno=-1)
```

Удалить диапазон страниц от *from_page* до *to_page*:

```
deletePageRange(from_page=5, to_page=10)
```

Скопировать ссылку на страницу под номером *pno* на позицию *to* (по умолчанию копирование осуществляется в конец документа):

```
copyPage(pno, to =-1 )
```

Для создания полной копии страницы нужно использовать следующую функцию:

```
fullcopyPage(pno , to =-1)
```

Перемещение страницы:

```
movePage(pno , to =-1)
```

Для закрытия файла используйте метод *Document.close()*.

Для сохранения – *Document.save()*.

Рассмотрим несколько примеров по редактированию PDF.

Извлечение страниц в виде изображений в формате PNG. Открываем исходный документ, в цикле осуществляем сохранение страницы (начиная с 11 и до 15, так как отсчёт страниц идёт с 0). Сгенерированные файлы изображений хранятся в каталоге с программой:

```
import fitz  
fname = "test.pdf"  
doc = fitz.open(fname)  
for page in doc.pages(10, 15, 1):  
    pix = page.getPixmap(alpha = False)  
    pix.writePNG("page-%i.png" % page.number)
```

Выполним обратное преобразование и соберём из изображений файл PDF. Для этого нам потребуется библиотека *os* для работы с директориями и файлами. Извлечём из папки *img* список файлом, их количество, затем в цикле будем конвертировать изображения в PDF (*img.convertToPDF*) и размещать их на страницах (*page.showPDFpag*):

```
import os, fitz  
  
doc = fitz.open()  
imgdir = "img"  
imglist = os.listdir(imgdir)  
imgcount = len(imglist)  
  
for i, f in enumerate(imglist):  
    img = fitz.open(os.path.join(imgdir, f))  
    rect = img[0].rect  
    pdfbytes = img.convertToPDF()  
    img.close()
```



```

imgPDF = fitz.open("pdf", pdfbytes)
page = doc.newPage(width=rect.width,
                    height=rect.height)
page.showPDFpage(rect, imgPDF, 0)
doc.save("res.pdf")

```

Следующий пример посвящён поиску текста в документе.

```

import fitz

def search_word(page, text):
    found = 0
    wlist = page.getTextWords()
    for w in wlist:
        if text in w[4]:
            found += 1
            r = fitz.Rect(w[:4])
            page.addUnderlineAnnot(r)
    return found

fname = "test.pdf"
text = "документ"
doc = fitz.open(fname)
new_doc = False
for page in doc:
    found = search_word(page, text)
    if found:
        new_doc = True
        print("Найдено", text, "(", found, " раз на странице №", (page.number +
1), ")")
if new_doc:
    doc.save("результат поиска-" + doc.name)

```

Функция *search_word* осуществляет поиск ключевого слова на каждой странице. В случае, если слово найдено, оно выделяется в документе подчёркиванием и рамкой. Если было найдено хотя бы одно вхождение, документ с отмеченными словами сохраняется отдельно.

Программа использует метод *Page.getTextWords()* для поиска строки, однако, он разделяет текст на отдельные слова, поэтому поисковый запрос не может содержать пробелы.

Последний пример – запись текста в документ. Поддержка кириллицы частичная.

```
import fitz

doc = fitz.open()
page = doc.newPage()
p = fitz.Point(50, 72)
text = """test
test 2
test 3"""
rc = page.insertText(p, text)
doc.save("test2.pdf")
```

Полная документация по данной библиотеке доступна по адресу:

<https://pymupdf.readthedocs.io/>

Отметим, что библиотека PyMuPDF предоставляет действительно широкие возможности по работе с документами формата PDF.

2.4. Работа с LaTeX на Python

LaTeX представляет собой систему подготовки высококачественной технической и научной документации. LaTeX не является текстовым процессором, как MS Word или LibreOffice. Философия LaTeX заключается в том, чтобы вместо того, чтобы тратить время на оформление и работу с интерфейсом, сосредоточиться на контенте и содержании текста. Перед тем как перейти к вопросу автоматизации работы с LaTeX в Python, рассмотрим структуру документа LaTeX. Он состоит из двух основных частей:

Преамбула: содержит подробную информацию о документе, такую как класс документа, имя автора, название и так далее.

Тело: основной текст, разделы, таблицы, математические уравнения, графики, рисунки и так далее.

Всё содержимое документа находится между операторами `'/begin{document}'` и `'/end{document}'`.

Отметим основные особенности LaTeX: возможность подготовки в едином формате отчётов, статей, книг, презентаций; контроль за оформлением объёмных документов с большим количеством разделов, рисунков, формул; формирование сложных математических формул; автоматическая генерация библиографии, индексов, ссылок.

Для создания и компиляции документов типа LaTeX мы будем использовать библиотеку PyLaTeX. С её помощью можно получить доступ ко всем функциям LaTeX в Python, создавать документы с меньшим количеством строк кода, а так как Python является языком высокого уровня, то проще написать код для PyLaTeX на Python по сравнению с LaTeX.

Рассмотрим создание документа. Прежде всего нужно осуществить подготовку для работы с LaTeX. Установите MikTeX (<https://miktex.org/download>) и модуль pylatex и импортируйте его в код Python.

В библиотеке содержатся объекты основных классов документов: статья, отчёт, письмо и так далее. Для того чтобы сформировать документ определённого класса, необходимо создать объект класса Document и в качестве аргумента передать нужный тип:

```
doc=Document(documentclass='article')
```

Далее импортируем стили и классы разделов:

```
from pylatex import Document, Section, Subsection  
from pylatex.utils import italic, bold
```

Для генерации PDF-файла документа используется метод `generate_pdf` класса Document, в качестве аргумента передаётся имя файла:

```
doc.generate_pdf("имя файла")
```

Рассмотрим пример создания простейшего документа:

```
from pylatex import Document, Section, Subsection, Tabular  
from pylatex import Math
```

```
doc = Document(documentclass="article",fontenc="T2A")
```

```
with doc.create(Section('Заголовок')):
```

```
doc.append('Текст')
```

```
doc.append('\nРазные символы: 234I?.,.@!${})')
```

```
with doc.create(Subsection('Формулы')):
```

```
doc.append(Math(data=['1+5', '=', 6]))
```

```
with doc.create(Subsection('Таблица')):
```

```
with doc.create(Tabular('r/c/c/l')) as table:
```

```
table.add_hline()
```

```
table.add_row((1, 2, 3, 4))
```

```
table.add_hline(1, 2)
```

```

table.add_empty_row()
table.add_row((4, 5, 6, 7))
table.add_row(("a", "b", "c", "d"))
doc.generate_pdf('full', clean_tex=False)

```

После выполнения программы формируется .tex файл, все необходимые сопроводительные файлы, а также компилируется PDF (рис. 2.6).

1 Заголовок

Текст

Разные символы: 2341?.,@!\${}

1.1 Формулы

$$1 + 5 = 6$$

1.2 Таблица

1	2	3	4
4	5	6	7
a	b	c	d

Рис. 2.6. Создание документа LaTeX в Python

Для успешной компиляции .tex файла в PDF необходимо выполнить следующие действия:

Установить Perl (ссылка для Windows: <http://strawberryperl.com/>)

В MikTeX (если Linux, то через консоль) установить пакет latexmk.

Для корректной работы с русским языком необходимо при создании документа указать аргумент fontenc="T2A". Этот пакет указывает внутреннюю кодировку в системе, поддерживающую русский язык.

Во втором примере добавим рисование графиков с использованием уже рассмотренной библиотеки matplotlib. Сформированный график будет помещён в объект Figure. Нумерация рисунка будет размещена автоматически.

```

import matplotlib
from pylatex import Document, Section, Figure, NoEscape
import matplotlib.pyplot as plt # noqa

matplotlib.use('Agg')
x = [0, 1, 2, 3, 4, 5, 6]
y = [15, 2, 7, 1, 5, 6, 9]
plt.plot(x, y)

```

```

doc = Document("test")
doc.append('Introduction.')
with doc.create(Section('Section')):
    doc.append('Plot:')
    with doc.create(Figure(position='htbp')) as plot:
        plot.add_plot(width=NoEscape(r'1\textwidth'))
        plot.add_caption('I am a caption.')
    doc.append('Created using matplotlib.')
doc.append('Conclusion.')
doc.generate_pdf(clean_tex=False)

```

Introduction.

1 Section

Plot:

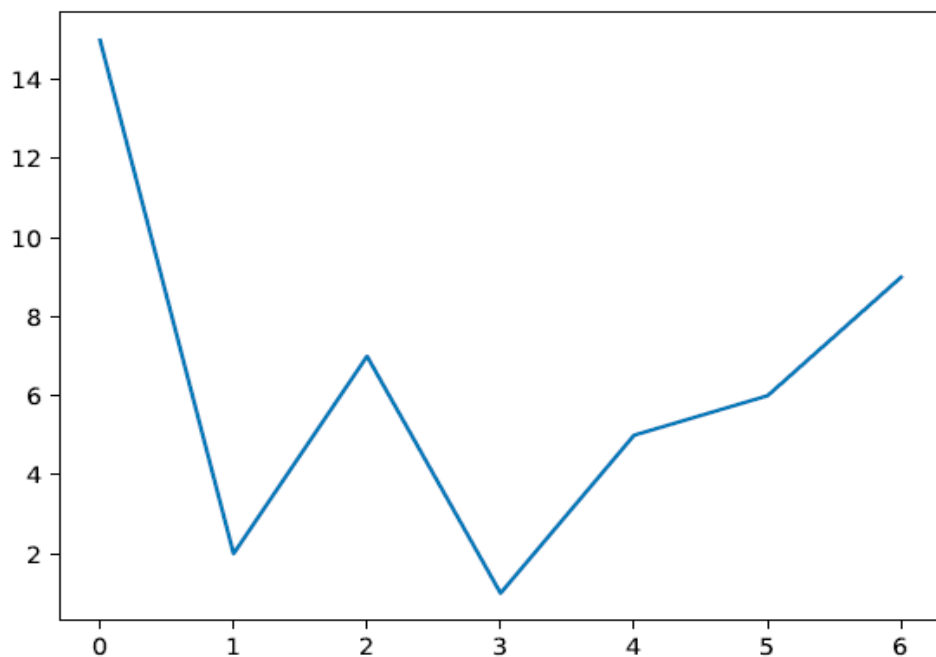


Figure 1: I am a caption.

Created using matplotlib.
Conclusion.

Рис. 2.7. Построение графика в LaTeX с помощью Mathplotlib

Полученный документ представлен на рис. 2.7. На его примере можно убедиться в способности Python благодаря множеству библиотек, способных работать совместно, осуществлять решение самых различных задач.

Наконец, в третьем примере мы рассмотрим списки и вставку рисунков.

```
import numpy as np
from pylatex import Document, Section, Subsection, Tabular, Math, TikZ, Axis, \
    Plot, Figure, Matrix, Alignat, Enumerate
import os

image_filename = os.path.join(os.path.dirname(__file__), 'test.png')
geometry_options = {"tmargin": "2cm", "lmargin": "10cm"}
doc = Document(geometry_options=geometry_options, fontenc="T2A")
a = np.array([[100, 10, 20]]).T
M = np.array([[2, 3, 4],
              [0, 0, 1],
              [0, 0, 2]])
with doc.create(Section('Секция')):
    doc.append('текст')
with doc.create(Section('Формулы')):
    with doc.create(Subsection('Матрицы')):
        doc.append(Math(data=[Matrix(M), Matrix(a), '=', Matrix(M * a)]))
    with doc.create(Subsection('Дробу')):
        with doc.create(Alignat(numbering=False, escape=False)) as agn:
            agn.append(r'\frac{a}{b} \&= 0 \\')
            agn.extend([Matrix(M), Matrix(a), '&=', Matrix(M * a)])
    with doc.create(Section('Списки')):
        with doc.create(Enumerate(enumeration_symbol=r"\alph*"),
                        options={'start': 20}) as enum:
            enum.add_item("the 1 item")
```

```

enum.add_item("the second item")
enum.add_item("the third etc \\ldots")
with doc.create(Subsection('зпафук')):
    with doc.create(TikZ()):
        plot_options = 'height=4cm, width=6cm, grid=major'
        with doc.create(Axis(options=plot_options)) as plot:
            plot.append(Plot(name='model', func='-x^5 - 242'))
            coordinates = [ (-4.77778, 2027.60977),
                            (-3.55556, 347.84069),
                            (-2.33333, 22.58953),
                            (-1.11111, -493.50066),
                            (0.11111, 46.66082),
                            (1.33333, -205.56286),
                            (2.55556, -341.40638),
                            (3.77778, -1169.24780),
                            (5.00000, -3269.56775),]
            plot.append(Plot(name='estimate', coordinates=coordinates))
        with doc.create(Subsection('рисунок')):
            with doc.create(Figure(position='h!')) as my_pic:
                my_pic.add_image(image_filename, width='120px')
                my_pic.add_caption('подпись')
doc.generate_pdf('full', clean_tex=False)

```

Результат выполнения этой программы представлен на рис. 2.8.

Таким образом, мы рассмотрели возможность применения Python для автоматизации формирования документов LaTeX. Естественно, рассмотренная библиотека имеет ряд недостатков, не позволяя реализовать полную функциональность редактора MikTeX. С другой стороны, ряд операций по подготовке и формированию текста, в особенности типовых документов, может быть осуществлён с помощью Python и библиотеки PyLaTeX.

1 Секция

текст

2 Формулы

2.1 Матрицы

$$\begin{pmatrix} 2 & 3 & 4 \\ 0 & 0 & 1 \\ 0 & 0 & 2 \end{pmatrix} \begin{pmatrix} 100 \\ 10 \\ 20 \end{pmatrix} = \begin{pmatrix} 200 & 300 & 400 \\ 0 & 0 & 10 \\ 0 & 0 & 40 \end{pmatrix}$$

2.2 Дроби

$$\frac{a}{b} = 0$$
$$\begin{pmatrix} 2 & 3 & 4 \\ 0 & 0 & 1 \\ 0 & 0 & 2 \end{pmatrix} \begin{pmatrix} 100 \\ 10 \\ 20 \end{pmatrix} = \begin{pmatrix} 200 & 300 & 400 \\ 0 & 0 & 10 \\ 0 & 0 & 40 \end{pmatrix}$$

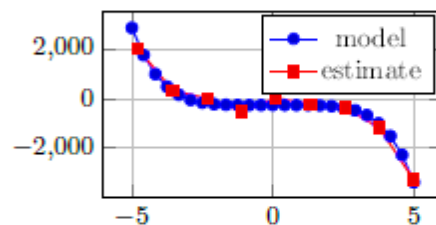
3 Списки

t) the 1 item

u) the second item

v) the third etc \ldots

3.1 график



3.2 рисунок

L^AT_EX

Figure 1: подпись

Рис. 2.8. Работа с графикой в LaTeX

Вопросы для закрепления

1. Какую структуру имеет документ формата .docx в библиотеке python-docx? Назовите основные функции и возможности этой библиотеки.
2. Приведите пример создания документа с рисунком.

3. Опишите работу с электронными таблицами с помощью библиотек `xlsxwriter` и `OpenPyXL`.

4. Приведите пример создания электронной таблицы с диаграммой.

5. Какие библиотеки Python используются для создания документов PDF?

6. Какие основные операции можно осуществить над документами PDF?

Приведите примеры на Python.

7. Что такое LaTeX? Для чего он используется?

8. Опишите основные возможности по работе с документами LaTeX в Python.

3. ОБРАБОТКА ИНФОРМАЦИИ В ОБЛАЧНЫХ СИСТЕМАХ

3.1. Работа с репозиториями

В рамках данного раздела мы рассмотрим работу с системой контроля версий Git и репозиторием GitHub. Использование данных инструментов не только повышает надёжность хранения проектов, но и обеспечивает комфортную коллективную работу над ними, удобную синхронизацию между различными устройствами.

Сначала рассмотрим основные команды и понятия.

Создание нового пустого репозитория:

```
git init
```

Проверка состояния репозитория, в том числе по текущей ветке:

```
git status
```

Ветка (branch) – это именное направление разработки, например, основная, тестовая, стабильная и т.д. По умолчанию создаётся ветка master. Изменение файлов в ветке, их переключение, объединение, удаление и другие операции подтверждаются коммитами (Commit).

Отслеживание файла, добавленного в директорию с репозиторием:

```
add <имя/маска файла>
```

Для удаления файла из списка отслеживания используется

```
git rm cached <имя/маска файла>
```

Чтобы внести изменения в репозиторий, его нужно подтвердить коммитом с указанием текста изменений:

```
git commit -m "Первая правка"
```

При первом запуске Git может потребовать регистрацию для идентификации пользователя, вносящего изменения. Для этого нужно выполнить две команды:

```
git config --global user.email "name@mail.ru"
```

```
git config --global user.name "Username"
```

У каждой версии в Git есть однозначно определяющий его идентификатор – хеш. Посмотреть историю версий можно с помощью команды *git log*.

Отменить изменения в файле, если они ещё не были подтверждены, можно следующей командой:

```
git checkout -- test.py
```

Чтобы зафиксировать изменения во всех файлах, используйте параметр -a:

```
git commit -a -m "Все сохранено"
```

Для переключения между версиями необходимо скопировать хеш коммита (или 4 и более первых символа) и вызвать команду

```
git checkout <хеш>
```

Для сравнения версий используйте

```
git diff <коммит 1> <коммит 2>
```

Далее рассмотрим работу с ветками. Каждый разработчик может вести свою ветку, после завершения работы объединять полученный результат с другими ветками, либо осуществлять тесты, не боясь испортить рабочий вариант.

Сначала создадим новую ветку

```
git checkout -b <имя ветки>
```

Список всех веток доступен через

```
git branch
```

Удаление ненужной ветки:

```
git branch -d <имя ветки>
```

Для объединения текущей ветки с какой-либо другой используется команда

```
git merge <имя ветки>
```

Для работы не только с локальным, но и удалённым репозиторием, его нужно подключить следующей командой:

```
git remote add <имя связи> <ссылка>
```

Например, создадим связь с репозиторием под именем `github1`:

```
git remote add github1 https://github.com/user/repository.git
```

Для отправки в удалённый репозиторий коммитов используется команда

```
git push -u <имя репозитория> <имя ветки>
```

По умолчанию команда `git push` работает с текущей активной веткой.

Если локального репозитория у вас нет и нужно организовать работу с удалённым, вызывается команда клонирования:

```
git clone <ссылка>
```

Для обновления изменений в скопированном локальном репозитории с удалённым используется команда обновления:

```
git fetch <имя связи>
```

Также имеется команда для получения изменений с сервера и автоматического слияния с локальной копией:

```
git pull <имя связи>
```

Ознакомившись с основными командами, далее рассмотрим организацию работы с Git средствами PyCharm.

Перед началом работы нужно убедиться, что PyCharm обнаружил установленный в системе Git. Для этого в меню настроек (File – Settings или Ctrl+Alt+S) нужно выбрать пункт Version Control – Git и проверить, что указан путь до исполняемого файла `git.exe`. Если нет, вам придётся указать его вручную.

Создадим проект на основе удалённого репозитория. Для этого на стартовом экране PyCharm нужно выбрать Check out from Version Control – Git. Авторизуйтесь и укажите нужный репозиторий.

Проект откроется как любой локальный, однако будут доступны дополнительные функции по работе с системой контроля версий во вкладке VCS, а также на панели:

- VCS – Commit содержит команды `git add`, `git commit` и `git push`.
- VCS – Update Project реализует `git pull`.

Остальные команды расположены в меню VCS – Git.

В окне совершения коммита (VCS – Commit) видны все файлы, которые были изменены по сравнению с предыдущей версией репозитория. Вы можете выбрать те файлы, что войдут в коммит. Также существует кнопка для отмены изменений к последней зафиксированной версии файла: Revert или Ctrl+Alt+Z.

Закоммитить изменения можно только после ввода сообщения. После этого можно либо сделать локальный коммит, либо сделать коммит и сразу отправить его на сервер (стрелочка справа на кнопке Commit открывает выпадающее меню, где можно выбрать пункт Commit and Push).

Посмотреть список удалённых репозиториев можно, выбрав пункт меню VCS – Git – Remotes. Здесь же можно добавить новую ссылку на удалённый репозиторий (зелёный «плюс» в правом верхнем углу).

Сравнить файл с последней зафиксированной версией – VCS – Git – Compare with the Same Repository Version. Сравнить последний коммит с любым другим можно с помощью VCS – Git – Compare with. Сравнить текущую ветку с другой поможет пункт VCS – Git – Compare with Branch (посмотреть текущую ветку и переключиться на другую можно в правом нижнем углу окна PyCharm).

Мы рассмотрели случай, когда удалённый репозиторий уже существует. PyCharm позволяет также создать новый локальный репозиторий для проекта и опубликовать его на GitHub. Создадим новый проект (File – NewProject) и добавим в него Python-файл. Чтобы добавить проект под систему контроля версий, нужно выбрать пункт меню VCS – Enable Version Control Integration, выбрать систему Git и нажать «ОК». После этого вы получите сообщение об успешном создании репозитория. Далее нужно обязательно сделать первый коммит.

Сначала добавим созданный файл к отслеживанию. Для этого в списке файлов кликаем по нему правой кнопкой мыши и выбираем пункт Git – Add (название файла должно измениться с красного на зелёный). После этого можно сделать первый коммит.

Далее разместим наш репозиторий на GitHub: выбираем пункт VCS – Import into Version Control – Share Project on GitHub. В открывшемся окне задаём имя репозитория на GitHub, а также имя для новой ссылки – можно оставить origin. Если всё прошло успешно, PyCharm сообщит об этом и покажет ссылку на опубликованный репозиторий.

В случае конфликтов между версиями PyCharm представляет удобный интерфейс для их ликвидации. В случае возникновения конфликтов в результате

обновления репозитория или объединения веток появляется всплывающее окно, в котором нужно нажать кнопку Merge. Новое окно будет разделено на три части: левая – текущая версия файла на ветке, в которую вливаются изменения; правая часть – версия файла на ветке, откуда вливаются изменения; центральная часть – версия, предлагаемая при попытке автоматического слияния. Также присутствуют кнопки для быстрого принятия изменений (Accept Left, Accept Right и Apply). Можно отменить слияние (Abort). Что выбрать, решает разработчик, также он может сам отредактировать центральную часть.

GitHub не является единственным решением для организации совместной работы. Большой популярностью пользуются также его конкуренты: GitLab, BitBucket, SourceForge и другие.

3.2. Реализация сетевых и облачных приложений на Python на основе Flask

Следующим этапом освоения облачных технологий с использованием Python будет разработка простейших веб-приложений. Python поддерживает возможность создания веб-страниц без установки дополнительных библиотек. Выглядит это следующим образом:

```
from http.server import HTTPServer, CGIHTTPRequestHandler
server_address = ("", 8000)
server1 = HTTPServer(server_address, CGIHTTPRequestHandler)
server1.serve_forever()
```

Этот код создаёт простейший веб-сервер, и если мы перейдём по адресу localhost:8000 (или 127.0.0.1:8000), то увидим список файлов директории, из которой запущена программа. В ней можно разместить набор HTML-страниц, мультимедиа-файлов и так далее. Однако для полноценной и удобной Web-разработки предпочтительнее использование более функциональных решений в виде готовых фреймворков, таких как Django и Flask. В данном пособии мы будем разбирать работу с Flask [11, 12].

Рассмотрим пример простейшей страницы. Сначала создаётся объект класса Flask, у которого вызывается метод *run* с атрибутами порта и адреса. Создание самих страниц осуществляется путём реализации функций с декораторами (функциями-обёртками, изменяющими поведение функции без изменения кода). В де-

коратор передаётся адрес (алиас) страницы, по которой она будет доступна. Таким образом, декоратор создаёт связь между адресом (URL) в браузере ('/' и '/index') и функцией *index()*. Информация, которая будет отображена на странице, формируется в функции и возвращается оператором *return*.

В данном примере страница отображает только текст, но позже мы рассмотрим, как формировать полноценные страницы с оформлением, формами, медиа-контентом и так далее.

```
from flask import Flask
app = Flask("name")
@app.route('/')
@app.route('/index')
def index():
    return "Привет"

app.run(port=8080, host='127.0.0.1')
```

Страницы веб-сайтов содержат не только текст, но и изображения, таблицы стилей, шрифты, файлы, музыку и видео. Все эти данные Flask ищет в директории *static*. Для строгой организации файлов рекомендуется создавать в *static* подпапки для каждого типа контента.

Рассмотрим пример страницы с картинкой, загруженной из папки *static*:

```
from flask import Flask, url_for
app = Flask("name")

@app.route('/image')
def image():
    return "<img src=\"{}\" alt=\"image\">".format(url_for('static', filename='img/test.png'))

app.run(port=8080, host='127.0.0.1')
```

Мы можем передавать на страницу определённые параметры (без использования GET-запроса). Для этого в декоратор помещается конструкция вида «*<тип:переменная>*», где тип принимает следующие значения:

- *string* – (по умолчанию) любой текст;
- *int* – положительное целое число;

- `float` – положительное вещественное число;
- `path` – строка со слешами;
- `uuid` – строка-идентификаторов из 16 байт в шестнадцатеричном представлении.

Рассмотрим пример страницы с передачей двух параметров: строки *username* и числа *number*:

```
@app.route('/params/<username>/<int:number>')
def two_params(username, number):
    return "<!doctype html><html lang='en'><head>
        <meta charset='utf-8'>
        <title>Пример</title>
        </head>
        <body><h2>{}</h2><div>1 параметр и его mun: {}</div>
        <h2>{}</h2><div>2 параметр и его mun: {}</div>
        </body></html>".format(username, str(type(username))[1:-1], number,
        str(type(number))[1:-1])
```

В примере выше мы использовали разметку прямо в коде. Такой подход нельзя назвать красивым или правильным. Предпочтительным вариантом является использование шаблонов с разметкой и интеграция в них данных. Таким образом отделяется frontend от backend-а. Во Flask шаблоны записываются как отдельные файлы, хранящиеся в папке `templates`. Добавим эту папку и файл с шаблоном для одной из страниц – `index.html` со следующим содержимым:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>{{ title }}</title>
</head>
<body>
<h1>Здравствуй, {{ username }}!</h1>
</body>
</html>
```


Для импорта данного шаблона используем функцию *render_template* из модуля flask:

```
@app.route('/')
@app.route('/index')
def index():
    user = "пользователь"
    return render_template('index.html', title='Главная страница',
username=user)
```

Функция *render_template* вызывает шаблонизатор Jinja2. Jinja2 заменяет блоки `{{...}}` на соответствующие им значения, переданные как аргументы шаблона. Рассмотрим ещё один пример, в котором будем использовать условные операторы:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Чётное - нечётное</title>
</head>
<body>
{% if number % 2 == 0 %}
<div>{{ number }} - чётное</div>
{% else %}
<div>{{ number }} - нечётное</div>
{% endif %}
</body>
</html>
```

Для интеграции шаблона в приложении создадим отдельный обработчик:

```
@app.route('/test')
def test():
    return render_template('odd_even.html', number=2)
```

В шаблонах также можно применять циклы:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Циклы в шаблонах</title>
</head>
<body>
{% for item in data%}
    <h2>{{item["name"]}}</h2>
    <div>{{item["content"]}}</div>
{% endfor %}
</body>
</html>
```

Обработчик событий для этой страницы будет иметь вид:

```
@app.route('/page2')
def page2():
    data = [{"name": "article 1", "content": "article content"},
            {"name": "article 2", "content": "none"}]
    return render_template('page2.html', data=data)
```

3.2.1. Работа с формами на Flask

Формы позволяют нам работать с интерактивными элементами, такими как:

- button – кнопка;
- checkbox – множественный выбор (флажки);
- color – поле выбора цвета;
- date, datetime, month, time, week – ввод даты и времени;
- email – поле для ввода адреса электронной почты;
- file – поле для выбора файла;
- number – поле для ввода числовой информации;
- password – поле для ввода пароля;
- radio – выбор одного из нескольких вариантов;

- `range` – ползунок для выбора значения из диапазона;
- `submit` – кнопка для отправки формы;
- `tel` – поле для ввода телефона;
- `text` – поле для ввода текста;
- `url` – поле для ввода адреса в Интернете.

Flask может значительно расширить свою функциональность за счёт дополнительных модулей. Далее рассмотрим работу с формами с использованием модуля `Flask-WTF`, позволяющего организовать объектно-ориентированный подход при взаимодействии с компонентами форм. Для защиты приложения от подделки запросов к форме настоятельно рекомендуется использовать следующую настройку (разместите её после создания переменной `app`):

```
app.config['SECRET_KEY'] = '1234567890key_!0000000000_abc'
```

По умолчанию `Flask-WTF` предотвращает любые варианты CSFR-атак. Это делается с помощью встраивания специального токена в скрытый элемент `<input>` внутри формы. Затем этот токен используется для проверки подлинности запроса. Для генерации этих токенов и задаётся секретный ключ – строка некоторой длины. Ключ должен быть надёжно защищён и быть достаточно длинным.

Создам класс простой формы, которую можно использовать для авторизации в приложении:

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import DataRequired

class LoginForm(FlaskForm):
    username = StringField('Логин', validators=[DataRequired()])
    password = PasswordField('Пароль', validators=[DataRequired()])
    remember_me = BooleanField('Запомнить меня')
    submit = SubmitField('Войти')
```

Из модуля `wtforms` импортируются основные элементы формы: поля, кнопки и т.д., а также набор валидаторов – методов, проверяющих, удовлетворяет ли состояние объекта формы определённому условию (не пустое, соответствие определённому типу, длине и т.д.). Для созданной формы также необходимо сформировать шаблон.

```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title</title>
</head>
<body>
<h1>Авторизация</h1>
<form action="" method="post" novalidate>
  {{ form.hidden_tag() }}
  <p>
    {{ form.username.label }}<br>
    {{ form.username }}<br>
    {% for error in form.username.errors %}
    <div class="alert alert-danger" role="alert">
      {{ error }}
    </div>
    {% endfor %}
  </p>
  <p>
    {{ form.password.label }}<br>
    {{ form.password }}<br>
    {% for error in form.password.errors %}
    <div class="alert alert-danger" role="alert">
      {{ error }}
    </div>
    {% endfor %}
  </p>
  <p>{{ form.remember_me() }} {{ form.remember_me.label }}</p>
  <p>{{ form.submit() }}</p>
</form>
</body>
</html>

```

Теперь мы работаем не с разметкой, а атрибутами объекта *form*. *form.hidden_tag* – атрибут, который добавляет в форму токен для защиты от атаки. Также добавлены циклы для обработки возможных ошибок (так как валидатор может быть неединственным). Закончим работу с формой, добавив обработчик страницы авторизации:

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        return redirect('/success')
    return render_template('login.html', title='Авторизация', form=form)
```

В обработчике мы создаём объект формы, проверяем, что все поля прошли валидацию и осуществляем в случае успеха переход на другую страницу (функцией *redirect*).

3.3. Реализация API и микросервисов на Python

При реализации веб-приложений можно выделить два подхода: монолитный и сервис-ориентированный. Монолитная архитектура веб-приложения включает три основные части: веб-сервер с логикой работы приложения, базу данных и пользовательский интерфейс. Недостатки монолитной архитектуры (например, необходимость развёртывания приложения целиком при любом изменении) привели к появлению модульной, а затем сервис-ориентированной архитектуры. Сервис-ориентированная архитектура основана на использовании отдельных полностью самостоятельных модулей (сервисов).

Внутри этого подхода можно выделить направление микросервисов, где приложение состоит из множества небольших сервисов, взаимодействующих между собой (чаще всего по протоколу HTTP). Каждый микросервис самостоятелен и независим, поэтому можно изменять только фрагменты приложения, не затрагивая остальных элементов, распределять сервисы по разным серверам, реализовывать их на разных языках программирования и с применением различных технологий.

На практике существует несколько стилей (стандартов) разработки приложения в соответствии с микросервисной идеологией. В рамках данного пособия мы рассмотрим подход REST (REpresentational State Transfer) [13].

REST – это архитектурный стиль программного обеспечения, включающий набор правил и ограничений на механику взаимодействия сервера и клиента в сети, основанный на использовании существующих стандартов: HTTP, URL, JSON, XML и других. Основным понятием в REST является URL. Для каждой единицы информации строится уникальный URL. Например, это может быть конструкция вида «/раздел/1/подраздел/2/объект/5/параметр_объекта/3» для получения 3 параметра объекта № 5 во втором подразделе первого раздела.

Для взаимодействия и управления информацией определены несколько протоколов передачи данных. REST использует протокол HTTP и осуществляет управление информацией с помощью HTTP-запросов: GET (для получения), PUT (для замены), POST (для добавления), DELETE (для удаления). Таким образом, можно выделить пять основных операций.

Основные виды HTTP-запросов.

Обозначение	Пример	Описание
GET	/items/5	Получить сведения о пятом объекте
GET	/items/	Получить сведения обо всех объектах
POST	/items/	Добавить новый объект
PUT	/items/5	Изменить пятый объект
DELETE	/items/5	Удалить пятый объект

Данные для добавления или изменения информации в запросах передаются посредством JSON или XML.

Приложения, поддерживающие архитектуру REST, называются RESTful-приложениями. Рассмотрим реализацию такого приложения на Python.

Создадим обычное приложение Flask и несколько обработчиков. В данном случае нам не потребуется создавать шаблоны, так как каждая функция будет отмечать лишь за получение, передачу или изменение информации, а ответ будет формироваться в виде JSON.

```
from flask import Flask, jsonify, request
app = Flask("test")
data = [1,2,3,4,5]
```

```

@app.route('/data/', methods=['GET'])
def get_data():
    return jsonify({'data': data})

@app.route('/data/<int:index>', methods=['GET'])
def get_one_data(index):
    if index<len(data):
        return jsonify({'data': data[index]})
    else:
        return jsonify({'error': "Wrong index"})

@app.route('/data/<int:index>', methods=['DELETE'])
def delete_data(index):
    if index<len(data):
        del(data[index])
        return jsonify({'success': 'OK'})
    else:
        return jsonify({'error': "Wrong index"})

@app.route('/data/<int:index>', methods=['PUT'])
def null_data(index):
    if index<len(data):
        data[index] = 0
        return jsonify({'success': 'OK'})
    else:
        return jsonify({'error': "Wrong index"})

@app.route('/data/', methods=['POST'])
def put_data():
    print(request.json)
    if not request.json:
        return jsonify({'error': 'Empty request'})
    elif 'data' not in request.json.keys() :
        return jsonify({'error': 'Bad request'})
    data.extend(request.json['data'])
    return jsonify({'success': 'OK'})

app.run(port=8088, host='127.0.0.1')

```

Дадим краткие пояснения к каждой функции:

get_data(): возвращает все элементы списка;

get_one_data(index): возвращает элемент списка под номером *index*, в случае выхода за рамки массива отправляется сообщение об ошибке (в качестве самостоятельной работы попробуйте переписать этот обработчик таким образом, чтобы проверка осуществлялась на основе обработки исключения *try..except*);

delete_data(index): удаление элемента с указанным индексом;

null_data(index): изменение элемента с указанным индексом;

put_data(): добавление нового элемента с проверкой ключа во входящем JSON.

Далее рассмотрим, как можно протестировать работу нашего API с помощью запросов. Все запросы осуществляются посредством библиотеки *requests*.

```
from requests import post, get, delete, put
```

```
print(get('http://localhost:8088/data/').json())  
print(post('http://localhost:8088/data/', json={'data': [9, 8, 7]}).json())  
print(post('http://localhost:8088/data/', json={'data2': [1, 2, 3]}).json())  
print(get('http://localhost:8088/data/').json())  
print(get('http://localhost:8088/data/0').json())  
print(get('http://localhost:8088/data/1').json())  
print(delete('http://localhost:8088/data/0').json())  
print(get('http://localhost:8088/data/').json())  
print(put('http://localhost:8088/data/1').json())  
print(get('http://localhost:8088/data/').json())
```

Часть представленных запросов будет вызывать ошибки, однако для тестирования вашего API необходимо проверять не только запросы в правильной форме и с правильным форматом данных, но и различные ошибочные случаи, чтобы проверить корректную обработку ошибок и выдачу соответствующих предупреждений и сообщений.

Представленные методы можно использовать и для работы со сторонними API, что мы рассмотрим далее.

3.4. Работа с API различных облачных сервисов

В заключение рассмотрим работу с некоторыми известными и популярными облачными сервисами, API которых доступно бесплатно [14].

3.4.1. Яндекс.Переводчик

Документация по использованию Яндекс.Переводчика расположена по следующему адресу:

<https://yandex.ru/dev/translate/doc/dg/concepts/about-docpage/>

Перед началом работы нужно получить бесплатный API-ключ. Он доступен по адресу: <https://translate.yandex.ru/developers/keys>. Нажмите кнопку «создать новый ключ», после чего добавьте описание и нажмите кнопку «создать». Скопируйте полученный ключ.

Рассмотрим пример работы с API-переводчика. В данном примере распознаётся язык введённого текста:

```
from requests import post, get

key = "<<<уникальный ключ>>>"
text = input()
params = {
    "key": key,
    "text": text
}

response = get("https://translate.yandex.net/api/v1.5/tr.json/detect",
params=params)
r_json = response.json()
print(r_json)
```

Во втором примере реализуем функцию перевода. Для выбора направления перевода (заданного парой кодов языков «с какого»-«на какой») используется атрибут *lang*. Например, «en-ru» обозначает перевод с английского на русский. Атрибут может содержать единственное значение, тогда он учитывается как конечный язык, а начальный определяется автоматически.

```

from requests import post, get

text = input()
lang = input()
params = {
    "key": key,
    "text": text,
    "lang": lang
}
response = get("https://translate.yandex.net/api/v1.5/tr.json/translate",
params=params)
r_json = response.json()
print(r_json)

```

Обратите внимание, что у бесплатного ключа есть лимит на количество переведённых символов.

3.4.2. Яндекс.Алиса

Далее кратко рассмотрим работу с API голосового помощника Яндекс.Алиса. Данный ассистент умеет распознавать естественную речь, имитирует живой диалог, даёт ответы на вопросы пользователя. Благодаря реализации тысяч запрограммированных навыков, Алиса способна решать множество прикладных задач. Рассмотрим разработку таких навыков (приложений).

Создание навыков (сервисов) осуществляется по адресу:

<https://yandex.ru/dev/dialogs/alice/>

Документация расположена по адресу:

<https://yandex.ru/dev/dialogs/alice/doc/about-docpage/>

Работа с Алисой происходит по технологии WebHook, идея которой заключается в реализации своего API по заданным правилам, после чего доступ к нему предоставляется Алиса для самостоятельного взаимодействия. Передача информации осуществляется посредством JSON.

Рассмотрим компоненты JSON-запроса. Со стороны пользователя поступают следующие данные:

request – данные, полученные от пользователя, включающие следующие элементы: *command* – запрос, переданный вместе с командой активации навыка; *original_utterance* – полный текст запроса;

session – информация о сессии, включает следующие атрибуты: *message_id* – идентификатор сообщения в рамках сессии; *session_id* – идентификатор сессии; *skill_id* – идентификатор вызываемого навыка; *user_id* – идентификатор пользователя в приложении, через которое он взаимодействует с Алисой;

new – флаг новой сессии;

version – версия протокола.

Со стороны Алисы в ответ поступает следующий JSON:

response – ответ пользователю, включающий следующие элементы: *text* – отображаемый для пользователя текст; *buttons* – массив кнопок, содержащий текст кнопки *title*, ссылку *url* и признак *hide*, означающий, что кнопку нужно скрыть после следующей команды пользователя;

end_session – признак конца разговора;

session – данные о сессии (аналогичные предыдущему запросу);

version – версия протокола.

Текст запроса, поступивший от пользователя, анализируется встроенной в Алису подсистемой поиска именованных сущностей. Она находит имена, фамилии, названия городов, время, место и т.д., позволяя упростить работу по распознаванию текста и его обработке. Распознанные сущности автоматически попадают в специальный раздел JSON *nlu*, включающий следующие компоненты:

tokens – обозначение начала и конца именованной сущности в массиве слов;

start – первое слово именованной сущности;

end – последнее слово после именованной сущности;

type – тип именованной сущности;

value – формальное описание именованной сущности, например имя.

Рассмотрим простейший пример, доступный в документации Яндекс.Алисы. Для запуска навыка рекомендуем осуществить следующую подготовку:

1. Зарегистрироваться на сервисе <https://www.pythonanywhere.com/>. Бесплатный аккаунт позволяет разместить одно приложение Flask.

2. Создать приложение Flask, нажав на «Add a new web app», выбрать последние версии Flask и Python.

3. Перейти на вкладку Files и открыть папку, в которую вы разместили Flask (по умолчанию mysite).

4. Заменить код в файле flask_app.py на код вашего приложения.

5. Перезапустить приложение (это необходимо делать после каждого изменения в коде) через вкладку Web и кнопку Reload Сайт.pythonanywhere.com.

6. Ваше приложение будет доступно по адресу Сайт.pythonanywhere.com.

7. Создаём новый навык Алисы в сервисе Яндекс.Диалогов, вводим описание и необходимые сведения.

8. Адрес Сайт.pythonanywhere.com необходимо вставить в поле Webhook URL.

9. Далее сохраняем изменения и переходим во вкладку «Тестирование», где мы можем проверить работу навыка без его модерации.

Далее представлен пример простейшего навыка из документации к Алисе.

```
from flask import Flask, request
import logging
import json

app = Flask(__name__)
logging.basicConfig( level=logging.INFO)
sessionStorage = {}

@app.route('/post', methods=['POST'])
def main():
    logging.info('Request: %r', request.json)
    response = {
        'session': request.json['session'],
        'version': request.json['version'],
        'response': { 'end_session': False }
    }
    handle_dialog(request.json, response)
    logging.info('Response: %r', request.json)
    return json.dumps(response)
```

```

def handle_dialog(req, res):
    user_id = req['session']['user_id']
    if req['session']['new']:
        sessionStorage[user_id] = {'suggests': ["Нem.", "He бyду.", "Xвaтum"]}
        res['response']['text'] = 'Привет! Купи слона!'
        res['response']['buttons'] = get_suggests(user_id)
        return

    if req['request']['original_utterance'].lower() in ['ладно', 'куплю', 'хорошо']:
        res['response']['text'] = 'Слона можно найти на Яндекс.Маркете!'
        res['response']['end_session'] = True
        return

    res['response']['text'] = 'Все говорят'
    "+str(req['request']['original_utterance'])+", а ты купи слона!'
    res['response']['buttons'] = get_suggests(user_id)

def get_suggests(user_id):
    session = sessionStorage[user_id]
    suggests = [{'title': suggest, 'hide': True} for suggest in session['suggests'][:2]]
    session['suggests'] = session['suggests'][1:]
    sessionStorage[user_id] = session
    if len(suggests) < 2:
        suggests.append({
            "title": "Ладно",
            "url": "https://market.yandex.ru/search?text=слон",
            "hide": True
        })
    return suggests

if __name__ == '__main__':
    app.run()

```

Дадим некоторые пояснения. Функция *main* принимает запрос и возвращает ответ. В *request.json* хранится запрос от Алисы. В *response* – запрос пользователя, который мы отправляем Алисе.

Схема работает как цикл, где условием остановки является `'end_session'=True`. До тех пор мы поочерёдно принимаем запрос от Алисы и отправляем ей сообщение или команду пользователя, в ответ получаем новый запрос с информацией и так далее. Обработку каждой итерации цикла можно вести с помощью условий, статусов, что позволит реализовать дерево вопросов и ответов.

Основная работа по обработке сообщения от Алисы и записи ответного запроса происходит в отдельной функции `handle_dialog`. В ней создаётся новая сессия, если это требуется, записывается информация о новом пользователе и формируется словарь с кнопками, которые отобразит Алиса.

Если пользователь уже ведёт диалог (сессия не новая), то осуществляется сначала обработка его сообщения и поиск ключевых слов. Если такие слова не найдены. Из подсказок удаляется одно слово и вопрос задаётся снова. Это продолжается несколько раз, после чего пользователю выдаётся единственная кнопка, ведущая на страницу поиска «товара».

При разработке навыков для Алисы вы также не ограничены в использовании сторонних библиотек. Это могут быть библиотеки по обработке изображений, аудио, поисковые сервисы, системы на основе технологий машинного обучения и так далее. В качестве самостоятельной работы попробуйте интегрировать API Яндекс-Переводчика в навык Алисы.

Вопросы для закрепления

1. Назовите основные команды для работы с локальным репозиторием Git.
2. Как переключиться между двумя ветками в Git?
3. Как подключиться к удалённому репозиторию Github через консоль, а также через интерфейс PyCharm?
4. Как создать простейшее Web-приложение с помощью Flask?
5. Опишите работу с шаблонами во Flask.
6. Дайте примеры работы с формами во Flask.
7. Что такое микросервисы и REST?
8. Дайте пример RESTful-приложения на Flask.
9. Какие облачные сервисы вы знаете? Приведите пример работы с каким-либо из них.

ЗАКЛЮЧЕНИЕ

В рамках данного учебного пособия рассмотрены основные подходы к анализу и обработке информации в офисных и облачных технологиях. В качестве основного инструмента предлагается язык программирования Python.

В пособии рассмотрены основные понятия офисных и облачных технологий, системного анализа и обработки данных. В первой главе представлены основы языка Python: типы данных, операторы, коллекции, функции, объектно-ориентированное программирование. Полученные знания используются для решения прикладных задач по обработке, анализу и представлению информации с помощью библиотек Pandas и Mathplotlib. Данные библиотеки позволяют автоматизировать и упростить работу с большими объёмами данных, осуществить построение графиков и диаграмм, что может быть полезным в учебной и научной деятельности.

Вторая глава посвящена применению Python для автоматизации работы с офисными документами. Рассмотренные библиотеки позволяют осуществить создание, редактирование и чтение документов, электронных таблиц, файлов формата PDF и LaTeX. Представленные инструменты, методы и подходы направлены на реализацию программного обеспечения, упрощающего типовые операции над документами.

В третьей главе рассмотрены облачные технологии и использование Python для работы с ними. Сначала читатель знакомится с системами контроля версий и репозиториями, так как организовать эффективную командную работу без их использования в настоящее время невозможно. Далее на примере популярного фреймворка Flask языка Python рассматривается создание простейших Web-приложений, работа с сетевыми запросами, создание собственного API и обращение к сторонним облачным сервисам.

Пособие будет полезно как студентам и магистрантам направления «Информатика и вычислительная техника», так и специалистам в области информационных технологий, так как позволяет сформировать базовые знания в области анализа и обработки информации в сфере офисных и облачных систем.

СПИСОК ЛИТЕРАТУРЫ

1. **Лутц, М.** Изучаем Python / М. Лутц. – СПб. : Символ–Плюс, 2011. – 1280 с.
2. **Van Rossum G.** PEP 8: style guide for Python code / Van Rossum G., B. Warsaw, N. Coghlan // Python. org. – 2001. – Т. 1565.
3. **Златопольский, Д.** Основы программирования на языке Python / Д. Златопольский. – Litres, 2019.
4. **Чибирова, М. Э.** Анализ данных и регрессионное моделирование с применением языков программирования Python и R / М. Э. Чибирова // Научные записки молодых исследователей. – 2019. – № 2.
5. **Доля, П. Г.** Введение в научный Python / П. Г. Доля. – Харьков : Харьковский национальный университет, 2016. – 333 с.
6. **McKinney, W.** Python for data analysis: Data wrangling with Pandas, NumPy, and IPython / W. McKinney. – «O'Reilly Media, Inc.», 2012.
7. **Tosi, S.** Matplotlib for Python developers / S. Tosi. – Packt Publishing Ltd, 2009.
8. **Влацкая, И. В.** Оценка современных средств обработки электронных документов / И. В. Влацкая, А. В. Максименко // Наука и современность. – 2010. – № 2-2. – С. 314 – 318.
9. **Кизянов, А. О.** Импорт данных из файлов MS Excel с помощью языка программирования python / А. О. Кизянов // Постулат. – 2017. – № 8.
10. **Poore, G. M.** PythonTeX: reproducible documents with LaTeX, Python, and more / G. M. Poore // Computational Science & Discovery. – 2015. – Т. 8, № 1. – С. 014010.
11. **Гринберг, М.** Разработка веб-приложений с использованием Flask на языке Python / М. Гринберг ; пер. с англ. А. Н. Киселева. – М. : ДМК Пресс, 2014.
12. **Васильев, П. А.** Web-программирование на языке Python. Фреймворки django, Flask / П. А. Васильев // Наука, техника и образование. – 2016. – № 8(26).
13. **Hillar, G. C.** Building RESTful Python Web Services / G. C. Hillar. – Packt Publishing Ltd, 2016.
14. **Петин, В. А.** API Яндекс, Google и других популярных веб-сервисов. Готовые решения для вашего сайта / В. А. Петин. – БХВ-Петербург, 2012.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ	3
1. ОСНОВЫ АНАЛИЗА И ОБРАБОТКИ ИНФОРМАЦИИ С ПОМОЩЬЮ ЯЗЫКА ПРОГРАММИРОВАНИЯ PYTHON	4
1.1. Основные понятия системного анализа и обработки данных в информационных системах	4
1.2. Основы языка программирования Python	5
1.3. Функциональное и объектно-ориентированное программирование в Python	10
1.4. Работа с библиотеками в Python	14
1.5. Применение языка программирования Python для анализа и обработки данных	17
1.6. Визуализация данных на Python с помощью библиотеки Mathplotlib	21
2. ОБРАБОТКА ИНФОРМАЦИИ В ОФИСНЫХ СИСТЕМАХ	30
2.1. Обработка офисных документов DOCX на Python	30
2.2. Обработка таблиц XLSX на Python	33
2.3. Работа с PDF на Python	43
2.4. Работа с LaTeX на Python	48
3. ОБРАБОТКА ИНФОРМАЦИИ В ОБЛАЧНЫХ СИСТЕМАХ	56
3.1. Работа с репозиториями	56
3.2. Реализация сетевых и облачных приложений на Python на основе Flask	60
3.3. Реализация API и микросервисов на Python	67
3.4. Работа с API различных облачных сервисов	71
ЗАКЛЮЧЕНИЕ	77
СПИСОК ЛИТЕРАТУРЫ	78

Учебное электронное издание

ОБУХОВ Артём Дмитриевич
КОРОБОВА Ирина Львовна

АНАЛИЗ И ОБРАБОТКА ИНФОРМАЦИИ В ОФИСНЫХ И ОБЛАЧНЫХ ТЕХНОЛОГИЯХ

Учебное пособие

Редактор Л. В. Комбарова

Инженер по компьютерному макетированию М. Н. Рыжкова

ISBN 978-5-8265-2174-8



Подписано к использованию 12.10.2020.

Тираж 50 шт. Заказ № 83

Издательский центр ФГБОУ ВО «ТГТУ»
392000, г. Тамбов, ул. Советская, д. 106, к. 14.
Телефон (4752) 63-81-08.
E-mail: izdatelstvo@tstu.ru

