Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Кузбасский государственный технический университет
имени Т. Ф. Горбачева»

Кафедра информатики и информационных систем

Составитель
Юлия Сергеевна Дементьева

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ

Методические материалы

Рекомендовано цикловой методической комиссией
Информационных систем и программирования
в качестве электронного издания
для использования в образовательном процессе

Кемерово 2026

Рецензенты:

Семенова О. С. – к. т. н., преподаватель кафедры информатики и информационных систем ФГБОУ ВО «Кузбасский государственный технический университет имени Т. Ф. Горбачева».

Дементьева Юлия Сергеевна
Основы алгоритмизации и программирования : методические материалы для обучающихся специальности СПО 09.02.11 Разработка и управление программным обеспечением / Кузбасский государственный технический университет имени Т. Ф. Горбачева, Кафедра информатики и информационных систем ; составитель Ю. С. Дементьева. – Кемерово : КузГТУ, 2026. – 1 файл (1205 КБ). – Текст : электронный.

Приведено содержание лабораторных и самостоятельных работ, а также материал, необходимый для успешного изучения дисциплины. Назначение издания – помощь обучающимся в получении знаний по дисциплине «Основы алгоритмизации и программирования» и организация практических работ.

© Кузбасский государственный
технический университет
имени Т. Ф. Горбачева, 2026
© Дементьева Ю.С., составление, 2026

ОГЛАВЛЕНИЕ

ЛАБОРАТОРНАЯ РАБОТА №1. ЛИНЕЙНЫЕ АЛГОРИТМЫ В PYTHON 5

ЛАБОРАТОРНАЯ РАБОТА №2. ПОСТРОЕНИЕ БЛОК-СХЕМ И ЗАПИСЬ АЛГОРИТМОВ В ПСЕВДОКОДЕ 10

ЛАБОРАТОРНАЯ РАБОТА №3. НАСТРОЙКА СРЕДЫ ПРОГРАММИРОВАНИЯ PYTHON И РАБОТА С IDE 15

ЛАБОРАТОРНАЯ РАБОТА №4. ИЗУЧЕНИЕ ИНТЕРПРЕТАТОРА PYTHON И ВЫПОЛНЕНИЕ ПРОСТЫХ ПРОГРАММ 18

ЛАБОРАТОРНАЯ РАБОТА №5. ТИПЫ ДАННЫХ И ОПЕРАЦИИ НАД НИМИ 22

ЛАБОРАТОРНАЯ РАБОТА №6. ВВОД-ВЫВОД ДАННЫХ И ОБРАБОТКА ИСКЛЮЧЕНИЙ 26

ЛАБОРАТОРНАЯ РАБОТА №7. ОПЕРАТОРЫ РАЗВЕТВЛЯЮЩИХСЯ ПРОГРАММ 29

ЛАБОРАТОРНАЯ РАБОТА №8. ЦИКЛИЧЕСКИЕ ПРОГРАММЫ НА PYTHON 32

ЛАБОРАТОРНАЯ РАБОТА №9. ПРАКТИЧЕСКИЕ ЗАДАЧИ НА ВЛОЖЕННЫЕ ЦИКЛЫ 35

ЛАБОРАТОРНАЯ РАБОТА №10. ОДНОМЕРНЫЕ МАССИВЫ И БАЗОВЫЕ ОПЕРАЦИИ 38

ЛАБОРАТОРНАЯ РАБОТА №11. ДВУМЕРНЫЕ МАССИВЫ 41

ЛАБОРАТОРНАЯ РАБОТА №12. СОРТИРОВКА МАССИВОВ 46

ЛАБОРАТОРНАЯ РАБОТА №13. ОБРАБОТКА МАССИВОВ: ПОИСК, ФИЛЬТРАЦИЯ, АГРЕГАЦИЯ 49

ЛАБОРАТОРНАЯ РАБОТА №14. СИМВОЛЫ И СТРОКИ. ОБРАБОТКА СТРОК 52

ЛАБОРАТОРНАЯ РАБОТА №15. ИСПОЛЬЗОВАНИЕ КОЛЛЕКЦИЙ 56

ЛАБОРАТОРНАЯ РАБОТА №16. РАБОТА С ФАЙЛАМИ. ЧТЕНИЕ И ЗАПИСЬ 59

ЛАБОРАТОРНАЯ РАБОТА №17. РАБОТА С КАТАЛОГАМИ И ФАЙЛАМИ 62

ЛАБОРАТОРНАЯ РАБОТА №18.ИСПОЛЬЗОВАНИЕ ПОДПРОГРАМ 66
ЛАБОРАТОРНАЯ РАБОТА №19.СЛОЖНЫЕ ФУНКЦИИ И РАЗЛИЧНЫЕ ТИПЫ ПАРАМЕТРОВ..... 69
ЛАБОРАТОРНАЯ РАБОТА №20.СОЗДАНИЕ МОДУЛЕЙ В PYTHON..... 72
ЛАБОРАТОРНАЯ РАБОТА №21.РАЗРАБОТКА РЕКУРСИВНЫХ ПОДПРОГРАММ..... 76
ЛАБОРАТОРНАЯ РАБОТА №22. РАБОТА С КЛАССАМИ. СОЗДАНИЕ КОНСТРУКТОРОВ..... 79
ЛАБОРАТОРНАЯ РАБОТА №23. ПРИМЕНЕНИЕ СВОЙСТВ И МЕТОДОВ 83
ЛАБОРАТОРНАЯ РАБОТА №24. НАСЛЕДОВАНИЕ..... 86
ЛАБОРАТОРНАЯ РАБОТА №25. ПОЛИМОРФИЗМ И ПЕРЕОПРЕДЕЛЕНИЕ МЕТОДОВ 89
ЛАБОРАТОРНАЯ РАБОТА №26. СОЗДАНИЕ ПРОЕКТА С КОМПОНЕНТАМИ ДЛЯ РАБОТЫ С ТЕКСТОМ 93
ЛАБОРАТОРНАЯ РАБОТА №27. КНОПОЧНЫЕ КОМПОНЕНТЫ И ОБРАБОТКА СОБЫТИЙ..... 97
ЛАБОРАТОРНАЯ РАБОТА №28. ИСПОЛЬЗОВАНИЕ ПЕРЕКЛЮЧАТЕЛЕЙ И ЭЛЕМЕНТОВ ВЫБОРА 101
ЛАБОРАТОРНАЯ РАБОТА №29. ОТОБРАЖЕНИЕ ТАБЛИЦ..... 104
ЛАБОРАТОРНАЯ РАБОТА №30. КОМПОНЕНТЫ ДЛЯ ОТОБРАЖЕНИЯ ДАТ И ВРЕМЕНИ 107
ЛАБОРАТОРНАЯ РАБОТА №31. РАЗРАБОТКА ИНТЕРФЕЙСА ПРИЛОЖЕНИЯ..... 110
ЛАБОРАТОРНАЯ РАБОТА №32. ТЕСТИРОВАНИЕ ПРИЛОЖЕНИЯ..... 114
Содержание самостоятельной работы 117
СПИСОК ЛИТЕРАТУРЫ 118

ЛАБОРАТОРНАЯ РАБОТА №1. ЛИНЕЙНЫЕ АЛГОРИТМЫ В PYTHON

Цель работы: изучить принципы построения линейных алгоритмов и их реализации на языке программирования Python; научиться описывать алгоритм в виде последовательности шагов, реализовывать его средствами языка Python без использования ветвлений и циклов, а также анализировать корректность полученных результатов.

Теоретическая часть

Алгоритм – это конечная последовательность строго определенных действий, выполнение которых приводит к решению поставленной задачи. Алгоритм должен обладать следующими свойствами: определенностью, конечностью, массовостью и результативностью.

По структуре управления алгоритмы делятся на три основных типа: линейные, разветвляющиеся и циклические. В данной лабораторной работе рассматриваются линейные алгоритмы как базовая форма организации вычислительного процесса.

Линейный алгоритм – это алгоритм, в котором все действия выполняются строго последовательно, одно за другим, без ветвлений и повторений. Порядок выполнения операторов соответствует порядку их записи.

Схематично линейный алгоритм можно представить следующим образом:

Начало → Ввод данных → Вычисления → Вывод результата → Конец

Каждая команда выполняется ровно один раз. Линейные алгоритмы используются при решении задач, не требующих проверки условий и повторения действий.

С помощью блок-схем линейный алгоритм может выглядеть следующим образом (см. рис. 1):

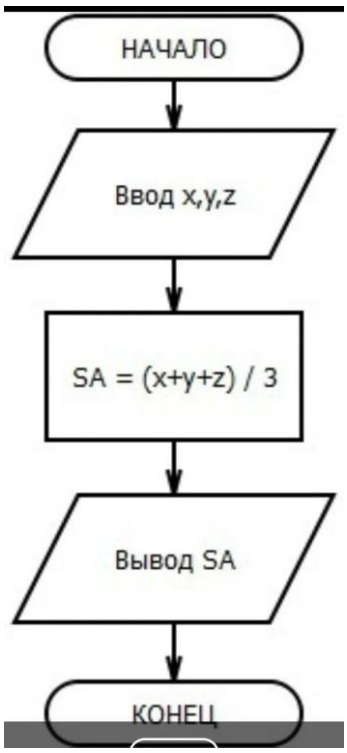


Рисунок 1. Линейный алгоритм

Python – интерпретируемый, объектно-ориентированный высокоуровневый язык программирования с динамической семантикой. Это высокоуровневый язык программирования общего назначения, ориентированный на повышение производительности разработчика и читаемости кода.

Установка Python на компьютер зависит от используемой операционной системы. Нужно понимать, что следует различать базовую установку, включающую интерпретатор, среду разработки IDLE, а также стандартную библиотеку, и установку дополнительных модулей, которых для Python написано очень много.

Программа на Python состоит из последовательности инструкций.

Основные элементы, используемые при реализации линейных алгоритмов:

Переменная – это именованная область памяти для хранения данных. В Python переменная создается при присваивании значения.

Пример:

a = 10

b = 5.2

Типы данных

В рамках лабораторной работы используются следующие типы данных:

1. None (неопределенное значение переменной).
2. Логические переменные (Boolean Type):
   - bool – логические значения – значения могут принимать одно из двух значений: True (истина) или False (ложь).
3. Числа (Numeric Type):
   - int – целое число – положительные и отрицательные целые числа, а также 0 (например, 4, 687, -45, 0);
   - float – число с плавающей точкой, вещественные (дробные) значения (например, 1.45, -3.789654, 0.00453). Для разделения целой и дробной частей здесь используется точка, а не запятая;
   - complex – комплексное число.
4. Списки (Sequence Type):
   - list – список,
   - tuple – кортеж,
   - range – диапазон.
5. Строки (Text Sequence Type):
   - str – символьная строка или единичный символ – набор символов, заключенных в кавычки (например, "ball", "Whatisyourname?", 'd', '6589'). Кавычки в Python могут быть одинарными или двойными.

Операции ввода и вывода

Ввод данных осуществляется при помощи оператора input():

a = input()

В скобках функции можно указать сообщение-комментарий к вводимым данным:

a = input ("Введите количество: ")

Вывод данных осуществляется при помощи оператора print():

```
a = 1
b = 2
print(a)
print(a + b)
print('сумма = ', a + b)
```

Внутри круглых скобок через запятую указываются те значения, которые нужно вывести.

По умолчанию функция print() принимает несколько аргументов, выводит их через пробел, после чего ставит перевод строки. Это поведение можно изменить, используя именованные параметры sep (разделитель) и end (окончание).

Для команды print может задаваться так называемый сепаратор – разделитель между элементами вывода:

x = 2
y = 5
print (x, "+", y, "=", x+y, sep = " ")

Результат отобразится с пробелами между элементами: 2 + 5 = 7

Арифметические операции

+ сложение,
- вычитание,
* умножение,
/ деление,
// целочисленное деление,
% остаток от деления,
** возведение в степень.

Требования к оформлению программы

Программа должна:

- содержать комментарии;
- иметь понятные имена переменных;
- корректно обрабатывать ввод вещественных чисел;
- выводить результат с поясняющим текстом;
- быть проверена на нескольких наборах входных данных.

Практическая часть

Задание 1. Напишите программу, которая выводит на первой строке «Привет, Python!», а на второй – «Приятно познакомиться!».

Задание 2. Напишите программу, которая считывает с клавиатуры последовательно три строки: название фильма, название кинотеатра и время, после чего выводит на экран «Билет на "[название фильма]" в "[название кинотеатра]" на [время] забронирован».

Задание 3. Ввести с клавиатуры три целых числа, найти их сумму, произведение и среднее арифметическое. Результат выводится с точностью до 3 знаков после запятой.

Задание 4. Напишите программу, которая запрашивает у пользователя следующие данные:

- ФИО ("Ваши фамилия, имя, отчество?"),
- возраст ("Сколько Вам лет?"),
- место жительства ("Где Вы живете?").

После этого программа должна вывести три строки:

"Ваше имя: "
"Ваш возраст: "
"Вы живете в: "
Возраст выводится с точностью до целых чисел.

Задание 5. Пользователь вводит число. Необходимо вывести;
- предыдущее число в экспоненциальной форме,
- текущее число в экспоненциальной форме,
- следующее число в экспоненциальной форме.

Контрольные вопросы

1. Дайте определение алгоритма.
2. Перечислите основные свойства алгоритма.
3. Какие типы алгоритмов выделяются по структуре управления?
4. Что представляет собой линейный алгоритм?
5. Какие типы данных используются в данной лабораторной работе?
6. Как осуществляется ввод данных в Python?
7. Как выполняется преобразование типов данных?
8. Перечислите основные арифметические операции.
9. Какие требования предъявляются к оформлению программы?
10. Почему важно тестировать программу на различных входных данных?

ЛАБОРАТОРНАЯ РАБОТА №2. ПОСТРОЕНИЕ БЛОК-СХЕМ И ЗАПИСЬ АЛГОРИТМОВ В ПСЕВДОКОДЕ

Цель работы: изучить способы представления алгоритмов в виде блок-схем и псевдокода; научиться формализовывать решение задачи до этапа программной реализации, разрабатывать графическую модель алгоритма с использованием стандартных обозначений, записывать алгоритм в форме псевдокода с соблюдением правил структурированного описания, а также анализировать логическую корректность построенной алгоритмической конструкции.

Теоретическая часть

Алгоритм является формализованным описанием процесса решения задачи. На этапе проектирования программного обеспечения особое значение приобретает представление алгоритма в наглядной и однозначной форме. Это позволяет выявить логические ошибки до начала программной реализации и обеспечивает структурированность вычислительного процесса.

Существуют различные способы представления алгоритмов:

- словесное описание;
- графическое представление (блок-схемы);
- табличная форма;
- запись в виде псевдокода;
- программная реализация на конкретном языке.

Наиболее распространенными способами предварительного описания алгоритма являются блок-схемы и псевдокод.

Блок-схема

Блок-схема – это графическая модель алгоритма, построенная с использованием стандартных графических символов, отражающих тип выполняемых действий. Блок-схемы позволяют наглядно представить последовательность выполнения операций и структуру управления алгоритмом.

Построение блок-схем регламентируется установленными правилами. Каждый элемент алгоритма изображается определенной геометрической фигурой (см. рис. 2).

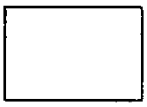
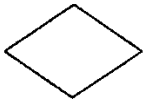
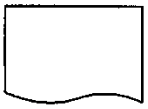
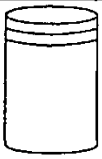
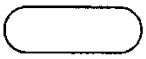
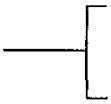
Наименование	Обозначение	Функции
Процесс		Выполнение операции или группы операций, в результате которых изменяется значение, форма представления или расположение данных.
Ввод-вывод		Преобразование данных в форму, пригодную для обработки (ввод) или отображения результатов обработки (вывод).
Решение		Выбор направления выполнения алгоритма в зависимости от некоторых переменных условий.
Предопределенный процесс		Использование ранее созданных и отдельно написанных программ (подпрограмм).
Документ		Вывод данных на бумажный носитель.
Магнитный диск		Ввод-вывод данных, носителем которых служит магнитный диск.
Пуск-останов		Начало, конец, прерывание процесса обработки данных.
Соединитель		Указание связи между прерванными линиями, соединяющими блоки.
Межстраничный соединитель		Указание связи между прерванными линиями, соединяющими блоки, расположенные на разных листах.
Комментарий		Связь между элементом схемы и пояснением.

Рисунок 2 - Условные обозначения блоков схем

Основные требования к построению блок-схем:

- однозначность интерпретации каждого блока;
- отсутствие «висящих» элементов без входа или выхода;
- соблюдение логической завершенности алгоритма;
- **аккуратность** и читаемость графического представления.

Блок-схемы применяются для анализа структуры алгоритма, выявления избыточных операций, проверки корректности логических ветвлений и циклов.

Псевдокод

Псевдокод – это формализованная текстовая запись алгоритма, не привязанная к синтаксису конкретного языка программирования.

Он представляет собой промежуточную форму между словесным описанием и программным кодом.

Псевдокод используется для:

- точной фиксации логики решения задачи;
- упрощения перехода к программной реализации;
- анализа корректности алгоритма;
- обсуждения алгоритма независимо от конкретной технологической среды.

Характерные особенности псевдокода:

- использование общеупотребительных ключевых слов (Начало, Конец, Если, Иначе, Пока и др.);
- запись действий построчно;
- применение операции присваивания (← или =);
- отсутствие строгих требований к синтаксису;
- логическая структурированность.

Пример записи разветвляющегося алгоритма в псевдокоде:

Начало
Ввести a, b
Если a > b
    Вывести a
Иначе
    Вывести b
Конец если
Конец

В псевдокоде важно соблюдать вложенность конструкций и логическую завершенность каждого блока. Это особенно существенно при описании разветвляющихся и циклических алгоритмов.

Практическая часть

В рамках выполнения лабораторной работы обучающемуся необходимо разработать блок-схемы и записать алгоритмы в форме псевдокода для предложенных задач.

Каждое задание должно содержать:

1. Краткое словесное описание алгоритма.
2. Блок-схему, выполненную с использованием стандартных обозначений.
3. Запись алгоритма в псевдокоде.

4. Краткий анализ корректности алгоритма (описание входных данных, ожидаемого результата, возможных граничных случаев).

Задание 1. Линейный алгоритм

Разработать алгоритм вычисления значения выражения:

S=(a+b)/2

Необходимо:

- Предусмотреть ввод значений a и b;
- Вывести результат вычисления;
- Построить блок-схему линейной структуры;
- Записать алгоритм в псевдокоде.

Задание 2. Разветвляющийся алгоритм

Разработать алгоритм определения наибольшего из трех чисел.

Необходимо:

- Ввести три числа a, b, c;
- Использовать вложенные условия;
- Вывести максимальное значение;
- Построить блок-схему с условными переходами;
- Записать алгоритм в псевдокоде.
- Дополнительно: предусмотреть ситуацию равенства чисел.

Задание 3. Циклический алгоритм

Разработать алгоритм вычисления суммы всех четных чисел от 1 до N.

Необходимо:

- Организовать ввод числа N;
- Использовать цикл (с предусловием или постусловием);
- Построить блок-схему циклической структуры;
- Записать алгоритм в псевдокоде;
- Проанализировать поведение алгоритма при $N \leq 0$.

Задание 4. Комбинированная структура

Разработать алгоритм проверки числа на простоту.

Необходимо:

- Использовать цикл и условный оператор;
- Предусмотреть корректную обработку чисел меньше 2;
- Построить блок-схему;
- Записать алгоритм в псевдокоде;
- Объяснить принцип работы алгоритма.

Контрольные вопросы

1. Что понимается под алгоритмом?
2. Какие существуют способы представления алгоритмов?
3. В чем преимущества графического представления алгоритма?
4. Какие основные блоки используются при построении блок-схем?
5. Какие требования предъявляются к построению блок-схем?
6. Что такое псевдокод?
7. Чем псевдокод отличается от программы на языке программирования?
8. Какие ключевые слова чаще всего используются в псевдокоде?
9. Что называется линейной структурой алгоритма?
10. Что такое разветвляющаяся структура?
11. Какие виды циклов существуют?
12. В чем различие между циклом с предусловием и циклом с постусловием?
13. Какие ошибки чаще всего допускаются при построении блок-схем?
14. Что понимается под логической корректностью алгоритма?
15. Зачем выполнять анализ алгоритма до этапа программной реализации?

ЛАБОРАТОРНАЯ РАБОТА №3. НАСТРОЙКА СРЕДЫ ПРОГРАММИРОВАНИЯ PYTHON И РАБОТА С IDE

Цель работы: изучить принципы установки и настройки среды программирования Python, а также освоить базовые приемы работы в интегрированной среде разработки. В ходе выполнения работы обучающийся должен научиться устанавливать интерпретатор Python, настраивать рабочее окружение, создавать и запускать программные файлы в IDE, анализировать сообщения об ошибках и корректно организовывать структуру проекта.

Теоретическая часть

Программирование на языке Python требует наличия установленного интерпретатора и программной среды, обеспечивающей удобство разработки и отладки кода.

Python относится к *интерпретируемым языкам программирования*. Это означает, что выполнение программы осуществляется построчно специальной программой – интерпретатором.

Интерпретатор выполняет следующие функции:

- чтение исходного кода;
- синтаксический анализ;
- выполнение инструкций;
- вывод сообщений об ошибках.

Для работы необходимо установить актуальную версию интерпретатора с официального сайта Python Software Foundation. При установке важно:

- выбрать версию Python 3;
- включить параметр добавления Python в системную переменную PATH;
- проверить корректность установки через командную строку (команда python --version).

Интегрированная среда разработки (IDE)

Интегрированная среда разработки (IDE – Integrated Development Environment) представляет собой программный комплекс, включающий редактор кода, средства запуска программ, отладчик и дополнительные инструменты.

Использование IDE обеспечивает:

- подсветку синтаксиса;
- автоматическое дополнение кода;
- поиск и исправление ошибок;
- удобную организацию файлов проекта;
- запуск и тестирование программ.

Для выполнения лабораторной работы может использоваться одна из следующих сред разработки:

- IDLE – стандартная среда, поставляемая вместе с Python;
- PyCharm – профессиональная среда разработки;
- Visual Studio Code – расширяемый редактор кода с поддержкой Python.

Основные элементы интерфейса IDE

Независимо от выбранной среды разработки, пользователь взаимодействует со следующими компонентами:

Редактор кода – область для ввода и редактирования программы.

Консоль (терминал) – область вывода результатов выполнения программы и сообщений об ошибках.

Проводник проекта – структура файлов и каталогов.

Панель инструментов – средства запуска и отладки.

Структура программного проекта

Программа на Python сохраняется в файле с расширением .py.

Рекомендуется соблюдать следующие правила организации проекта:

- создавать отдельную папку для каждой лабораторной работы;
- использовать осмысленные имена файлов;
- избегать пробелов и специальных символов в названиях;
- хранить исходные файлы и отчет в одной структуре каталогов.

При запуске программы возможны ошибки, которые можно условно разделить на: синтаксические ошибки (SyntaxError), ошибки выполнения (RuntimeError) и логические ошибки.

IDE позволяет определить строку, в которой возникла ошибка, и тип проблемы. Анализ сообщений об ошибках является важной частью процесса разработки.

Практическая часть

Задание 1. Установка и проверка Python

1. Установить интерпретатор Python версии 3.x.
2. Проверить корректность установки с помощью командной строки.
3. Сделать скриншот результата проверки версии.

Задание 2. Установка и настройка IDE

1. Установить выбранную среду разработки.
2. Создать новый проект «Лабораторная работа №3».
3. Создать файл main.py.
4. Настроить интерпретатор Python в параметрах проекта.

Задание 3. Создание и запуск программы

Создать программу, которая:

- выводит на экран текст «Среда программирования настроена успешно»;
- запрашивает имя пользователя;
- выводит приветственное сообщение с использованием введенного имени.

Пример ожидаемого результата работы программы:

Введите ваше имя: Иван

Здравствуйте, Иван!

Задание 4. Анализ ошибок

1. Преднамеренно допустить синтаксическую ошибку (например, пропустить закрывающую скобку).
2. Запустить программу.
3. Зафиксировать сообщение об ошибке.
4. Исправить ошибку и повторно выполнить запуск.

Контрольные вопросы

1. Что такое интерпретатор?
2. В чем отличие интерпретируемых языков от компилируемых?
3. Что представляет собой IDE?
4. Какие функции выполняет интегрированная среда разработки?
5. Для чего используется переменная PATH?
6. Какие типы ошибок могут возникать при выполнении программы?
7. Как определить строку, в которой возникла ошибка?
8. Почему важно правильно организовывать структуру проекта?

ЛАБОРАТОРНАЯ РАБОТА №4. ИЗУЧЕНИЕ ИНТЕРПРЕТАТОРА PYTHON И ВЫПОЛНЕНИЕ ПРОСТЫХ ПРОГРАММ

Цель работы: изучить принципы функционирования интерпретатора Python и освоить базовые приемы выполнения программ в интерактивном и файловом режимах. В ходе выполнения работы обучающийся должен научиться работать с интерактивной оболочкой Python, выполнять отдельные команды и программные фрагменты, запускать файлы с исходным кодом, анализировать результаты выполнения и интерпретировать сообщения об ошибках.

Теоретическая часть

Язык Python относится к интерпретируемым языкам программирования. Выполнение программ осуществляется с помощью специальной программы – интерпретатора.

Интерпретатор Python выполняет следующие функции:

- считывание исходного кода;
- пооперационное выполнение инструкций;
- проверку синтаксической корректности;
- вывод результатов и сообщений об ошибках.

Интерпретатор Python может функционировать в двух основных режимах:

- Интерактивный режим (REPL – Read–Evaluate–Print Loop).
- Режим выполнения файла.

Интерактивный режим запускается из командной строки вводом команды python или python3. После запуска появляется приглашение.

В этом режиме пользователь может вводить отдельные команды, которые выполняются немедленно. Результат вычисления выводится сразу после выполнения выражения.

Пример работы в интерактивном режиме:

```
5 + 3
8
x = 10
x * 2
20
```

Интерактивный режим удобен для:

- проверки отдельных выражений;

- тестирования фрагментов кода;
- изучения синтаксиса языка;
- выполнения простых вычислений.

Режим выполнения файла – специальный режим в котором программа записывается в текстовом файле с расширением .py и выполняется полностью после запуска.

Пример структуры файла:

```
print("Программа запущена")
a = 5
b = 7
print("Сумма:", a + b)
```

Запуск в таком режиме возможен из IDE или через командную строку командой python имя_файла.py.

При выполнении файла все инструкции выполняются последовательно сверху вниз.

Структура простой программы

Простая программа на Python может включать:

- комментарии;
- операции ввода данных;
- вычисления;
- вывод результатов.

Комментарий начинается со знака # и не влияет на выполнение программы.

Пример:

```
name = input("Введите имя: ")
print("Здравствуйте,", name)
```

Сообщения об ошибках

При выполнении программы возможны ошибки:

Синтаксические ошибки (SyntaxError) – возникают при нарушении правил записи команд.

Ошибки времени выполнения (например, ZeroDivisionError) – возникают в процессе выполнения программы.

Логические ошибки – программа выполняется, но результат некорректен.

Интерпретатор указывает строку, в которой обнаружена ошибка, и ее тип. Умение анализировать такие сообщения является необходимым навыком программиста.

Для выхода из интерактивного режима используются:

- команда exit();

- сочетание клавиш Ctrl + D (в Unix-подобных системах);
- сочетание клавиш Ctrl + Z и Enter (в Windows).

Практическая часть

Задание 1. Работа в интерактивном режиме

1. Запустить интерпретатор Python.
2. Выполнить следующие команды:
   - вычислить значение выражения 12 + 8 * 3;
   - определить переменную x = 15;
   - вычислить x ** 2;
   - определить тип значения переменной x с использованием функции type().
3. Зафиксировать результаты выполнения.

Задание 2. Создание и выполнение программного файла

1. Создать файл lab4_task.py.
2. Реализовать программу, которая:
   - запрашивает у пользователя два числа;
   - выводит их сумму, произведение и разность;
   - сопровождает вывод поясняющим текстом.
3. Выполнить программу через IDE.
4. Выполнить программу через командную строку.

Задание 3. Анализ ошибок

1. Преднамеренно изменить программу, допустив ошибку (например, использовать неинициализированную переменную).
2. Запустить программу.
3. Зафиксировать сообщение об ошибке.
4. Проанализировать причину возникновения ошибки.
5. Исправить программу.

Контрольные вопросы

1. Что такое интерпретатор?
2. В чем заключается принцип работы интерпретируемого языка?
3. Какие режимы работы поддерживает интерпретатор Python?
4. В чем различие между интерактивным режимом и режимом выполнения файла?
5. Как создается переменная в Python?
6. Какие виды ошибок могут возникать при выполнении программы?

7. Как завершить работу в интерактивном режиме?
8. Почему интерактивный режим удобен при изучении языка?

ЛАБОРАТОРНАЯ РАБОТА №5.
ТИПЫ ДАННЫХ И ОПЕРАЦИИ НАД НИМИ

Цель работы: изучить основные встроенные типы данных языка Python и освоить операции, выполняемые над ними. В ходе выполнения работы обучающийся должен научиться определять тип данных переменной, выполнять арифметические и строковые операции, осуществлять преобразование типов, а также анализировать результаты вычислений с учетом особенностей представления данных в памяти.

Теоретическая часть

В языке Python каждая переменная имеет определенный тип данных. Тип данных определяет множество допустимых значений и набор операций, которые могут быть выполнены над этими значениями.

Python относится к языкам с динамической типизацией. Это означает, что тип переменной определяется автоматически при присваивании значения и может изменяться в процессе выполнения программы.

Основные встроенные типы данных

В рамках данной лабораторной работы рассматриваются следующие базовые типы данных:

1. **int** – целые числа, используются для хранения целочисленных значений: 10, −5, 0, 12456.
   Пример:
   a = 25
2. **float** – вещественные числа предназначены для хранения чисел с плавающей точкой: 3.14, −0.5, 2.0.
   Пример:
   b = 7.8

   Следует учитывать, что вещественные числа хранятся с ограниченной точностью.
3. **str** – строки используются для хранения текстовой информации. Строка заключается в одинарные или двойные кавычки.
   Пример:
   name = "Иван"
4. **bool** – логический тип, который может принимать два значения: True или False. Используется при проверке условий.

Определение типа данных

Для определения типа переменной используется функция type().

Пример:

```
x = 10
print(type(x))
```

Операции над числовыми типами

Для типов int и float доступны арифметические операции:

- Сложение +;
- Вычитание -;
- Умножение *;
- Деление /;
- Целочисленное деление //;
- Остаток от деления %;
- Возведение в степень **.

Результат операции может иметь тип, отличный от типа операндов. Например, при делении двух целых чисел оператором / результатом будет число типа float.

Операции над строками

Для строк доступны следующие операции:

1. Конкатенация (объединение строк) с помощью оператора +.
   Пример:
   ```
   "Привет, " + "мир"
   ```
2. Повторение строки с помощью оператора *.
   Пример:
   ```
   "abc" * 3
   ```
3. Определение длины строки с помощью функции len().

Строки являются неизменяемым типом данных. Это означает, что после создания строки ее содержимое нельзя изменить, можно только создать новую строку.

Преобразование типов

Для явного преобразования типов используются функции:

- int() – преобразование к целому числу;
- float() – преобразование к вещественному числу;
- str() – преобразование к строке;
- bool() – преобразование к логическому типу.

Пример:

```
x = "15"
y = int(x)
```

Ошибки преобразования возникают при попытке преобразовать строку, не содержащую числового значения, к типу int или float.

При выполнении операций важно учитывать совместимость типов. Например, попытка сложения строки и числа без явного преобразования приведет к ошибке типа TypeError.

Пример некорректной операции:

"Возраст: " + 20

Для корректного выполнения требуется преобразование:

"Возраст: " + str(20)

Практическая часть

Задание 1. Определение типов

1. Создать переменные различных типов (int, float, str, bool).
2. Определить тип каждой переменной с помощью функции type().
3. Зафиксировать результаты.

Задание 2. Арифметические операции

1. Ввести два целых числа.
2. Выполнить все основные арифметические операции над ними.
3. Определить тип результата каждой операции.
4. Сделать вывод о влиянии операторов на тип результата.

Задание 3. Работа со строками

1. Ввести имя и фамилию пользователя.
2. Объединить их в одну строку.
3. Определить длину полученной строки.
4. Вывести строку, повторенную два раза.

Задание 4. Преобразование типов

1. Ввести числовое значение в виде строки.
2. Преобразовать его к типу int и float.
3. Выполнить арифметические операции с полученными значениями.
4. Проанализировать возможные ошибки преобразования.

Контрольные вопросы

1. Что такое тип данных?
2. Какие базовые типы данных используются в Python?
3. В чем заключается динамическая типизация?
4. Какие арифметические операции поддерживаются для числовых типов?
5. Какие операции допустимы для строк?

6. Что такое конкатенация?
7. Какие функции используются для преобразования типов?
8. Какие ошибки могут возникнуть при несоответствии типов?
9. В чем различие между типами int и float?

ЛАБОРАТОРНАЯ РАБОТА №6.
ВВОД-ВЫВОД ДАННЫХ И ОБРАБОТКА ИСКЛЮЧЕНИЙ

Цель работы: изучить механизмы ввода и вывода данных в языке Python, а также освоить основы обработки исключений. В ходе выполнения работы обучающийся должен научиться организовывать корректный ввод данных с клавиатуры, форматировать вывод результатов, выявлять и обрабатывать ошибки выполнения программы с использованием конструкции обработки исключений.

Теоретическая часть

Организация ввода и вывода данных является неотъемлемой частью любой программы. В языке Python для взаимодействия с пользователем используются встроенные функции.

Ввод данных осуществляется с помощью функции input().

Синтаксис:

input(сообщение)

Функция возвращает введенное значение в виде строки (тип str). Для выполнения арифметических операций требуется явное преобразование типа с использованием функций int() или float().

Пример:

```
a = int(input("Введите целое число: "))
```

Следует учитывать, что при вводе некорректного значения (например, текста вместо числа) возникает ошибка преобразования.

Вывод данных осуществляется с помощью функции print().

Синтаксис:

print(значение1, значение2, ...)

Функция print() автоматически преобразует переданные аргументы в строковый формат.

Форматированный вывод

Для более удобного представления данных используются:

1. Конкатенация строк.
2. Форматирование с помощью метода format().
3. f-строки (форматированные строковые литералы).

Пример f-строки:

```
name = "Иван"
print(f"Здравствуйте, {name}")
```

Форматированный вывод позволяет управлять представлением чисел, например, ограничивать количество знаков после запятой.

Исключения

Исключение – это событие, возникающее во время выполнения программы и нарушающее ее нормальный ход.

В Python предусмотрен механизм обработки исключений с использованием конструкции try–except.

Синтаксис:

```
try:
    блок_кода
except ТипОшибки:
    блок_обработки
```

Пример:

```
try:
    x = int(input("Введите число: "))
except ValueError:
    print("Ошибка: введено некорректное значение")
```

Основные типы исключений:

1. ValueError – возникает при некорректном преобразовании типов данных.
2. ZeroDivisionError – деление на ноль.
3. TypeError – несовместимость типов данных при выполнении операции.
4. NameError – обращение к несуществующей переменной.

Также возможно использование блока *finally*, который выполняется независимо от возникновения ошибки.

Пример:

```
try:
    a = int(input("Введите число: "))
    b = int(input("Введите число: "))
    print(a / b)
except ZeroDivisionError:
    print("Деление на ноль недопустимо")
except ValueError:
    print("Ошибка ввода данных")
finally:
    print("Завершение программы")
```

Обработка исключений позволяет повысить устойчивость программы и предотвратить ее аварийное завершение.

Практическая часть

Задание 1. Организация корректного ввода

Разработать программу, которая:

- запрашивает у пользователя два вещественных числа;
- выводит их сумму, разность и произведение;
- форматирует вывод до двух знаков после запятой.

Задание 2. Обработка ошибок преобразования

Модифицировать программу таким образом, чтобы:

- при вводе некорректного значения выводилось сообщение об ошибке;
- программа не завершалась аварийно.

Задание 3. Обработка деления на ноль

Разработать программу, которая:

- запрашивает два числа;
- выполняет деление первого числа на второе;
- обрабатывает возможную ошибку деления на ноль;
- выводит корректное диагностическое сообщение.

Задание 4. Комплексное задание

Создать программу «Калькулятор», которая:

- запрашивает два числа;
- выполняет одну из операций (+, −, *, /);
- обрабатывает ошибки ввода;
- обрабатывает деление на ноль;
- выводит результат в форматированном виде.

Контрольные вопросы

1. Какая функция используется для ввода данных в Python?
2. В каком типе возвращается значение функцией input()?
3. Какие способы форматирования вывода существуют?
4. Что такое исключение?
5. Для чего используется конструкция try–except?
6. Какие типы исключений вам известны?
7. Что происходит при отсутствии обработки исключения?
8. Для чего используется блок finally?
9. Почему обработка ошибок является важной частью разработки программ?

ЛАБОРАТОРНАЯ РАБОТА №7. ОПЕРАТОРЫ РАЗВЕТВЛЯЮЩИХСЯ ПРОГРАММ

Цель работы: изучить принципы построения разветвляющихся алгоритмов и освоить операторы условного перехода в языке Python. В ходе выполнения работы обучающийся должен научиться использовать логические выражения для принятия решений, реализовывать конструкции if, elif, else, анализировать корректность условий и оценивать влияние ветвления на выполнение программы.

Теоретическая часть

Разветвляющийся алгоритм – это алгоритм, в котором выполнение дальнейших действий зависит от выполнения некоторого условия. В отличие от линейных алгоритмов, где команды выполняются строго последовательно, в разветвляющихся алгоритмах возможны альтернативные пути выполнения.

Структура разветвляющегося алгоритма может быть представлена следующим образом:

Начало → Проверка условия → Да → Действие 1 → Нет → Действие 2 → Конец

Для реализации разветвляющихся алгоритмов в языке Python используется оператор условного перехода if.

Оператор if

Базовая форма записи:

```
if условие:
    блок_команд
```

Блок команд выполняется только в том случае, если условие истинно (True).

Расширенная форма:

```
if условие:
    блок_1
else:
    блок_2
```

В данном случае при истинности условия выполняется блок_1, при ложности – блок_2.

Множественный выбор:

```
if условие_1:
    блок_1
elif условие_2:
```

блок_2

else:

блок_3

Конструкция elif используется для проверки дополнительных условий.

Логические выражения

Условие в операторе if представляет собой логическое выражение. В логических выражениях используются:

Операторы сравнения:

- Больше >
- Меньше <
- Больше или равно >=
- Меньше или равно <=
- Равно =
- Не равно !=

Логические операторы:

- and – логическое «и»
- or – логическое «или»
- not – логическое отрицание

Результатом логического выражения является значение типа bool: True или False.

Пример:

```
x = 10
if x > 0:
print("Число положительное")
```

Вложенные условные операторы

Оператор if может содержать внутри себя другие условные конструкции. Такие конструкции называются вложенными.

Пример:

```
if x > 0:
    if x % 2 == 0:
        print("Число положительное и четное")
```

При использовании вложенных операторов необходимо строгоо соблюдать правила отступов, так как в Python именно отступы определяют структуру программы и принадлежность блоков кода к той или иной ветке условия.

Практическая часть

Задание 1. Определение знака числа

Разработать программу, которая:

- запрашивает у пользователя число;
- определяет, является ли число положительным, отрицательным или равным нулю;
- выводит соответствующее сообщение.

Задание 2. Проверка четности

Разработать программу, которая:

- запрашивает целое число;
- определяет, является ли число четным или нечетным;
- выводит результат.

Задание 3. Нахождение максимального из трех чисел

Разработать программу, которая:

- запрашивает три числа;
- определяет наибольшее из них;
- выводит результат.

Задание 4. Классификация оценки

Разработать программу, которая:

- запрашивает количество набранных баллов (0–100);
- определяет оценку по следующей шкале:
  - 90–100 – «Отлично»,
  - 75–89 – «Хорошо»,
  - 60–74 – «Удовлетворительно»,
  - менее 60 – «Неудовлетворительно»;
- выводит соответствующее сообщение.

Контрольные вопросы

1. Что такое разветвляющийся алгоритм?
2. Для чего используется оператор if?
3. В чем различие между if и if–else?
4. Для чего применяется elif?
5. Какие операторы сравнения используются в Python?
6. Какие логические операторы используются в Python?
7. Что является результатом логического выражения?
8. Почему в Python важны отступы?
9. Какие ошибки чаще всего возникают при использовании условных операторов?

ЛАБОРАТОРНАЯ РАБОТА №8. ЦИКЛИЧЕСКИЕ ПРОГРАММЫ НА PYTHON

Цель работы: изучить принципы построения циклических алгоритмов и освоить операторы цикла в языке Python. В ходе выполнения работы обучающийся должен научиться реализовывать циклы с предусловием и с параметром, управлять процессом повторения действий, использовать счетчики и аккумуляторы, а также анализировать корректность работы циклических конструкций.

Теоретическая часть

Циклический алгоритм – это алгоритм, в котором одна или несколько команд выполняются многократно. Повторение действий осуществляется до выполнения определенного условия либо заданное количество раз.

В языке Python реализованы два основных вида циклов:

- цикл с предусловием (while);
- цикл с параметром (for).

Цикл while

Цикл while выполняет блок команд до тех пор, пока условие истинно.

Синтаксис:

```
while условие:
    блок_команд
```

Перед каждой итерацией проверяется условие. Если оно принимает значение True, выполняется тело цикла. Если False – выполнение цикла прекращается.

Пример:

```
i = 1
while i <= 5:
    print(i)
    i += 1
```

В данном примере используется счетчик i, значение которого изменяется на каждой итерации.

Важно учитывать, что при отсутствии изменения условия цикл может стать бесконечным.

Цикл for

Цикл for используется для перебора элементов последовательности или выполнения блока команд заданное количество раз.

Синтаксис:

for переменная *in* последовательность:

блок_команд

Чаще всего используется функция range(), которая генерирует последовательность чисел.

Формы записи range():

- range(n) – от 0 до n−1
- range(a, b) – от a до b−1
- range(a, b, step) – от a до b−1 с шагом step

Пример:

```
for i in range(1, 6):
    print(i)
```

Цикл выполнится 5 раз.

Вспомогательные операторы управления циклом:

- break – досрочное завершение цикла.
- continue – переход к следующей итерации, минуя оставшуюся часть тела цикла.

Пример:

```
for i in range(1, 6):
    if i == 3:
        continue
    print(i)
```

Переменные-счетчики и аккумуляторы

1. Счетчик – переменная, определяющая количество выполненных итераций.
2. Аккумулятор – переменная, накапливающая результат (например, сумму).

Пример:

```
summa = 0
for i in range(1, 6):
    summa += i
```

Здесь переменная summa является аккумулятором.

Практическая часть

Задание 1. Вывод последовательности чисел

Разработать программу, которая:

- выводит числа от 1 до 20 с использованием цикла for;
- выводит числа от 20 до 1 с использованием цикла while.

Задание 2. Вычисление суммы

Разработать программу, которая:

- запрашивает натуральное число n;
- вычисляет сумму чисел от 1 до n;
- выводит результат.

Задание 3. Таблица умножения

Разработать программу, которая:

- выводит таблицу умножения для числа, введенного пользователем (от 1 до 10).

Задание 4. Поиск числа

Разработать программу, которая:

- генерирует последовательность чисел от 1 до 50;
- завершает выполнение при достижении числа, кратного 7 и 5 одновременно;
- выводит найденное число.

Контрольные вопросы

1. Что такое циклический алгоритм?
2. В чем различие между циклами while и for?
3. Как работает функция range()?
4. Что такое бесконечный цикл?
5. Для чего используется оператор break?
6. Для чего используется оператор continue?
7. Что такое переменная-аккумулятор?
8. Какие ошибки чаще всего возникают при работе с циклами?
9. Почему важно правильно формулировать условие завершения цикла?

ЛАБОРАТОРНАЯ РАБОТА №9.
ПРАКТИЧЕСКИЕ ЗАДАЧИ НА ВЛОЖЕННЫЕ ЦИКЛЫ

Цель работы: изучить принципы построения вложенных циклов в языке Python, научиться решать задачи, требующие многократного повторения внутри повторения, а также формировать навыки работы с табличными и двумерными структурами данных.

Теоретическая часть

Вложенный цикл – это цикл, размещенный внутри другого цикла. При этом внешний цикл управляет количеством повторений внутреннего цикла.

Общая схема:

```
for i in range(n):
    for j in range(m):
        действия
```

Если внешний цикл выполняется n раз, а внутренний – m раз, то общее количество выполнений тела внутреннего цикла равно $n \times m$.

В языке Python вложенность не ограничена строго, однако чрезмерное количество уровней усложняет понимание программы и может привести к ошибкам.

Принцип работы вложенных циклов

Сначала выполняется первая итерация внешнего цикла. Затем полностью выполняется внутренний цикл. После его завершения внешний цикл переходит к следующей итерации, и внутренний цикл запускается снова.

Это важно понимать: внутренний цикл всегда выполняется полностью для каждой итерации внешнего цикла.

Типичные области применения вложенных циклов:

- построение таблиц;
- работа с двумерными массивами (матрицами);
- генерация узоров и фигур;
- перебор пар значений;
- реализация алгоритмов сравнения элементов.

Пример вывода таблицы умножения:

```
for i in range(1, 6):
    for j in range(1, 6):
        print(i * j, end=" ")
    print()
```

Здесь внешний цикл задает строку таблицы, внутренний – элементы в строке.

Вложенные циклы и сложность алгоритма

Количество операций растет пропорционально произведению числа итераций. Например, двойной цикл по 100 элементов выполнит 10 000 операций. Это важно учитывать при обработке больших объемов данных.

Практическая часть

Задание 1. Таблица квадратов

Разработать программу, которая:

- выводит таблицу квадратов чисел от 1 до 10;
- оформляет вывод в виде таблицы 10×2 (число и его квадрат).

Задание 2. Полная таблица умножения

Разработать программу, которая:

- выводит таблицу умножения от 1 до 10;
- оформляет вывод в виде квадратной таблицы.

Задание 3. Геометрические фигуры

Разработать программу, которая:

- выводит прямоугольник из символов * размером n × m (размеры вводит пользователь);
- выводит треугольник с основанием n;
- выводит «шахматный» узор из символов # и . размером n × n.

Задание 4. Подсчет пар чисел

Разработать программу, которая:

- перебирает все пары чисел от 1 до 20;
- выводит только те пары (a, b), для которых a + b кратно 7;
- подсчитывает количество таких пар.

Задание 5. Работа с матрицей

Создать двумерный список (матрицу) размером 3 × 3.

Программа должна:

- вывести матрицу;
- вычислить сумму элементов каждой строки;
- вычислить сумму элементов каждого столбца;
- найти максимальный элемент матрицы.

Контрольные вопросы

1. Что такое вложенный цикл?

2. Как выполняются внутренний и внешний циклы?
3. Сколько раз выполнится внутренний цикл при n итерациях внешнего?
4. Где применяются вложенные циклы?
5. Почему важно правильно выбирать диапазоны?
6. Что такое алгоритмическая сложность в контексте вложенных циклов?
7. Какие ошибки наиболее характерны при их использовании?

ЛАБОРАТОРНАЯ РАБОТА №10. ОДНОМЕРНЫЕ МАССИВЫ И БАЗОВЫЕ ОПЕРАЦИИ

Цель работы: изучить способы хранения и обработки набора однотипных данных в языке Python, освоить базовые операции над одномерными массивами (списками), научиться выполнять поиск, сортировку, модификацию и анализ элементов массива.

Теоретическая часть

Одномерный массив – это упорядоченная структура данных, предназначенная для хранения набора элементов одного типа. В языке Python роль одномерного массива выполняет структура данных **«список» (list)**.

Список – это изменяемая последовательность объектов. Элементы списка нумеруются, начиная с нуля.

Пример создания списка:

numbers = [5, 8, 12, 3, 7]

Доступ к элементам осуществляется по индексу:

- numbers[0] – первый элемент;
- numbers[2] – третий элемент.

Отрицательные индексы позволяют обращаться к элементам с конца:

- numbers[-1] – последний элемент.

Основные операции со списками

Основные операции со списками приведены в таблице 1.

Таблица 1 – Основные операции со списками

Операция	Описание	Пример
x in a	Проверка, содержится ли x в списке a	5 in [2, 3, 5]
x not in a	Проверка, что x не содержится в a	5 not in [2, 3, 6]
a + a2	Конкатенация списков (создается новый список: сначала элементы a, затем a2)	[2, 4] + [5, 3] == [2, 4, 5, 3]
a * k	Повторение списка a k раз	[2, 3] * 3 == [2, 3, 2, 3, 2, 3]
a[n]	Элемент списка с индексом n (отрицательные – с конца)	[2, 3, 7][0] == 2 [2, 3, 7][-1] == 7
a[start:stop:step]	Срез списка	[2, 3, 7][:2] == [2, 3]
len(a)	Длина списка	len([2, 3, 7]) == 3
max(a)	Максимальный элемент	max([2, 3, 7]) == 7
min(a)	Минимальный элемент	min([2, 3, 7]) == 2
sum(a)	Сумма элементов	sum([2, 3, 7]) == 12
a.index(x)	Индекс первого вхождения x (ошибка, если элемента нет)	[2, 3, 7].index(7) == 2

Операция	Описание	Пример
a.count(x)	Количество вхождений x	[2, 7, 3, 7].count(7) == 2
a.append(x)	Добавить x в конец списка	a = [2, 3, 7] a.append(8) → [2, 3, 7, 8]
a.extend(a2)	Добавить элементы a2 в конец a	a = [2, 3, 7] a.extend([8, 4, 5]) → [2, 3, 7, 8, 4, 5]
del a[n]	Удалить элемент с индексом n	a = [2, 3, 7] del a[1] → [2, 7]
del a[start:stop:step]	Удалить элементы по срезу	a = [2, 3, 7] del a[:2] → [7]
a.clear()	Очистить список	a.clear()
a.copy()	Копия списка	b = a.copy()
a += a2	Заменить содержимое на a + a2	a += [4, 5]
a *= k	Заменить содержимое на a * k	a *= 2
a.insert(n, x)	Вставить x на позицию n	a = [2, 3, 7] a.insert(0, 8) → [8, 2, 3, 7]
a.pop(n)	Получить и удалить элемент с индексом n (без аргумента – последний)	a = [2, 3, 7] a.pop(1) == 3 → [2, 7]
a.remove(x)	Удалить первое вхождение x	a = [2, 3, 7] a.remove(3) → [2, 7]
a.reverse()	Развернуть список	a = [2, 3, 7] a.reverse() → [7, 3, 2]
a.sort()	Сортировка по возрастанию	a = [3, 2, 7] a.sort() → [2, 3, 7]
a.sort(reverse=True)	Сортировка по убыванию	a = [3, 2, 7] a.sort(reverse=True) → [7, 3, 2]
bool(a)	Проверка на пустоту (True – если список непустой)	bool([1, 2]) == True bool([]) == False

Практическая часть

Задание 1. Ввод и вывод массива

Разработать программу, которая:

- запрашивает у пользователя количество элементов;
- формирует список из введенных чисел;
- выводит список на экран;
- выводит количество элементов.

Задание 2. Анализ массива

Для введенного массива определить:

- сумму элементов;
- среднее арифметическое;
- максимальный и минимальный элементы;
- количество четных чисел.

Задание 3. Модификация массива

Разработать программу, которая:

- удаляет из списка все отрицательные числа;
- заменяет все четные элементы на значение 0;
- выводит измененный список.

Задание 4. Поиск элемента

Разработать программу, которая:

- запрашивает число для поиска;
- определяет, присутствует ли оно в массиве;
- выводит индекс найденного элемента или сообщение об отсутствии.

Задание 5. Сортировка

Разработать программу, которая:

- сортирует массив по возрастанию;
- сортирует массив по убыванию;
- выводит оба результата.

Контрольные вопросы

1. Что такое одномерный массив?
2. Чем список в Python отличается от массива в других языках?
3. С какого индекса начинается нумерация элементов?
4. Что такое срез?
5. Чем отличается метод sort() от функции sorted()?
6. Какие ошибки возможны при обращении к элементам массива?
7. Почему не рекомендуется изменять список во время его перебора?

ЛАБОРАТОРНАЯ РАБОТА №11.
ДВУМЕРНЫЕ МАССИВЫ

Цель работы: изучение способов хранения и обработки двумерных массивов в языке Python, освоение базовых операций с матрицами, включая доступ к элементам, изменение значений, перебор строк и столбцов, а также выполнение вычислений над элементами массива.

Теоретическая часть

Двумерный массив – это структура данных, представляющая собой таблицу, состоящую из строк и столбцов. В Python двумерный массив реализуется с помощью списка списков. Каждый элемент внешнего списка соответствует строке матрицы, а элементы внутреннего списка – столбцам этой строки.

Создание двумерного массива:

```
matrix = [
[1, 2, 3],
[4, 5, 6],
[7, 8, 9]
]
```

Доступ к элементам:

- matrix[0][1] – элемент первой строки, второго столбца (значение 2)
- matrix[2][2] – элемент третьей строки, третьего столбца (значение 9)

Перебор элементов матрицы:

```
for row in matrix:
    for element in row:
        print(element, end=" ")
    print()
```

Или перебор элементов можно выполнить через индексы:

```
for i in range(len(matrix)):
    for j in range(len(matrix[i])):
        print(matrix[i][j], end=" ")
    print()
```

Размер матрицы

Для работы с двумерными массивами важно уметь определять их размеры. В Python двумерный массив реализуется как список списков, поэтому количество строк матрицы определяется длиной

внешнего списка, а количество столбцов – длиной внутреннего списка, соответствующего конкретной строке.

Например, если матрица задана как:

```
matrix = [
[1, 2, 3],
[4, 5, 6],
[7, 8, 9]
]
```

то количество строк вычисляется как len(matrix), что в данном случае равно 3, а количество столбцов в i-й строке – len(matrix[i]). Для первой строки это будет len(matrix[0]) = 3. Такое обращение позволяет динамически работать с матрицами любых размеров.

Базовые операции с матрицей

1. **Изменение элемента**

Для изменения значения отдельного элемента используется присваивание по индексу строки и столбца. Например, чтобы изменить элемент, расположенный во второй строке и третьем столбце, необходимо написать:

```
matrix[1][2] = 10
```

После выполнения этой команды значение элемента matrix[1][2] изменится на 10.

2. **Добавление строки или столбца**

Добавление строки: новая строка добавляется в конец матрицы с помощью метода append().

Пример:

```
matrix.append([10, 11, 12])
```

После этого матрица будет содержать четыре строки.

Добавление столбца: чтобы добавить столбец, необходимо для каждой строки добавить новый элемент с помощью метода append() или вставки insert().

Пример:

```
for row in matrix:
    row.append(0)  # добавляем в каждую строку новый элемент со значением 0
```

Таким образом, все строки матрицы увеличатся на один столбец.

3. **Удаление строки или столбца**

Удаление строки: используется оператор del, указывающий индекс удаляемой строки:

```
del matrix[1]  # удаляется вторая строка
```

Удаление столбца: удаление столбца выполняется путем удаления элемента с нужным индексом в каждой строке:

```
for row in matrix:
    del row[2]  # удаляется третий столбец
```

Важно соблюдать аккуратность при работе с индексами, чтобы не выйти за пределы массива.

4. **Сумма всех элементов**

Для подсчета суммы всех элементов используется вложенный цикл и аккумулятор:

```
total = 0
for row in matrix:
    for element in row:
        total += element
```

После выполнения этих команд переменная total будет содержать сумму всех элементов матрицы.

5. **Сумма элементов строки или столбца**

Сумма строки: отдельный цикл для выбранной строки:

```
row_sum = sum(matrix[0])  # сумма элементов первой строки
```

Сумма столбца: перебор всех строк и суммирование элементов с нужным индексом:

```
col_sum = 0
for row in matrix:
    col_sum += row[1]  # суммирование элементов второго столбца
```

6. **Максимальный и минимальный элемент**

Максимальный и минимальный элементы можно найти с помощью функций max() и min() для каждой строки, либо используя вложенный перебор всех элементов:

```
max_val = matrix[0][0]
for row in matrix:
    for element in row:
    if element > max_val:
        max_val = element
```

Аналогично вычисляется минимальное значение.

Типичные ошибки при работе с матрицами

1. Выход за пределы индекса: попытка обратиться к несуществующей строке или столбцу приводит к ошибке IndexError.

2. Путаница с индексами строк и столбцов: легко перепутать i и j при вложенных циклах.
3. Неправильный перебор элементов: использование неправильных диапазонов или шагов цикла может привести к пропуску элементов.
4. Изменение массива во время перебора: добавление или удаление строк/столбцов во время циклов может вызвать некорректное выполнение программы.

Правильное понимание структуры и индексации двумерного массива позволяет безопасно выполнять все необходимые операции и минимизировать ошибки при работе с матрицами.

Практическая часть

Задание 1. Ввод и вывод матрицы

1. Запросить у пользователя размеры матрицы (строк и столбцов).
2. Ввести элементы матрицы.
3. Вывести матрицу в виде таблицы.

Задание 2. Сумма и среднее арифметическое элементов

1. Вычислить сумму всех элементов матрицы.
2. Определить среднее арифметическое всех элементов.
3. Вывести результаты.

Задание 3. Минимальный и максимальный элемент

1. Найти максимальный элемент матрицы.
2. Найти минимальный элемент.
3. Вывести результаты.

Задание 4. Суммы строк и столбцов

1. Вычислить сумму каждого столбца и строки матрицы.
2. Вычислить сумму каждого столбца.
3. Вывести результаты.

Задание 5. Трансформация матрицы

1. Создать новую матрицу, в которой все отрицательные элементы заменены на 0.
2. Удвоить все элементы нечетных столбцов.
3. Вывести измененную матрицу в виде таблицы.

Контрольные вопросы

1. Что такое двумерный массив?
2. Как в Python реализуется двумерный массив?
3. Как обратиться к элементу матрицы по индексу?

4. Как определить количество строк и столбцов?
5. Как вычислить сумму всех элементов матрицы?
6. Какие ошибки чаще всего возникают при работе с матрицами?
7. Как изменить элемент матрицы?
8. Как добавить строку или столбец в матрицу?
9. Как удалить строку или столбец?

ЛАБОРАТОРНАЯ РАБОТА №12. СОРТИРОВКА МАССИВОВ

Цель работы: изучить методы сортировки одномерных массивов (списков) в языке Python, освоить стандартные функции и алгоритмы сортировки. Сформировать навыки выбора оптимального способа упорядочивания данных, а также провести анализ эффективности различных методов сортировки.

Теоретическая часть

Сортировка – это процесс упорядочивания элементов массива по возрастанию или убыванию. Упорядоченные данные облегчают поиск, анализ и обработку информации.

В Python одномерный массив представлен списком (list), и сортировка может быть выполнена несколькими способами:

1. **С помощью встроенных функций:**
   - sort() – метод списка, изменяющий исходный список на месте.
   - sorted() – функция, создающая новый отсортированный список, оставляя исходный без изменений.

   Примеры:

numbers = [5, 2, 8, 1, 9]
numbers.sort() # numbers станет [1, 2, 5, 8, 9]

numbers2 = [5, 2, 8, 1, 9]
sorted_numbers = sorted(numbers2) # numbers2 остается [5, 2, 8, 1, 9]
sorted_numbers – [1, 2, 5, 8, 9]

Обе функции поддерживают параметр reverse=True для сортировки по убыванию.

Алгоритмы сортировки

Помимо встроенных средств, полезно понимать базовые алгоритмы:

- **Пузырьковая сортировка (Bubble Sort)** – последовательное сравнение соседних элементов с обменом при необходимости.
- **Сортировка выбором (Selection Sort)** – поиск минимального (или максимального) элемента и перемещение его в начало массива.
- **Сортировка вставками (Insertion Sort)** – вставка элемента на правильное место среди уже отсортированных элементов.

Простейший пример пузырьковой сортировки:

```
for i in range(len(numbers)-1):
    for j in range(len(numbers)-i-1):
    if numbers[j] > numbers[j+1]:
    numbers[j], numbers[j+1] = numbers[j+1], numbers[j]
```

Анализ эффективности

Встроенная сортировка Python (Timsort) оптимизирована для большинства случаев.

Простые алгоритмы, такие как пузырьковая сортировка или сортировка вставками, полезны для понимания принципов, но для больших массивов менее эффективны ($O(n^2)$).

Практическая часть

Задание 1. Сортировка числового массива

1. Запросить у пользователя количество элементов массива.
2. Ввести элементы массива.
3. Отсортировать массив по возрастанию с помощью sort().
4. Вывести отсортированный массив.
5. Отсортировать массив по убыванию с помощью sorted() и вывести результат.

Задание 2. Сортировка строк

1. Запросить у пользователя список строк (например, имена).
2. Отсортировать строки в алфавитном порядке.
3. Отсортировать строки по длине каждого слова.
4. Вывести результаты.

Задание 3. Реализация пузырьковой сортировки

1. Реализовать алгоритм пузырьковой сортировки для числового массива.
2. Вывести массив до сортировки и после.
3. Сравнить результат с использованием встроенного метода sort().

Задание 4. Сортировка с ключами

1. Создать список словарей, например, студентов с полями «имя» и «оценка».
2. Отсортировать список по оценке (по убыванию).
3. Отсортировать список по имени (по алфавиту).
4. Вывести результаты.

Контрольные вопросы

1. Что такое сортировка массива?

2. Чем отличаются методы sort() и sorted()?
3. Какие алгоритмы сортировки вам известны?
4. Для чего используется параметр reverse?
5. Как реализуется пузырьковая сортировка?
6. Какие ошибки чаще всего возникают при ручной сортировке массива?
7. Почему встроенные методы Python более эффективны для больших массивов?

ЛАБОРАТОРНАЯ РАБОТА №13. ОБРАБОТКА МАССИВОВ: ПОИСК, ФИЛЬТРАЦИЯ, АГРЕГАЦИЯ

Цель работы: освоить методы обработки одномерных массивов в языке Python, научиться выполнять поиск элементов, фильтрацию данных по заданным условиям и вычисление агрегированных характеристик массива.

Теоретическая часть

Одномерный массив в языке Python представлен списком (list). Работа с массивами редко ограничивается простым хранением данных. Чаще требуется найти нужные элементы, отобрать данные по определенным критериям или вычислить обобщенные показатели.

Поиск в массиве

Поиск – это процесс нахождения элемента по значению или условию.

1. **Линейный поиск**

   Последовательный перебор всех элементов массива до нахождения нужного значения.

   Пример:

```
numbers = [4, 7, 1, 9, 3]
target = 9
found = False
for i in range(len(numbers)):
    if numbers[i] == target:
    print("Найдено на позиции:", i)
    found = True
    break
```

   Если элемент не найден, цикл завершится без срабатывания условия.

2. **Поиск по условию**

   Например, поиск первого четного числа или первого элемента больше заданного значения.

Фильтрация массива

Фильтрация – это отбор элементов массива, удовлетворяющих заданному условию.

Пример: формирование нового списка из четных чисел:

```
even_numbers = []
```

for num in numbers:

if num % 2 == 0:

even_numbers.append(num)

Также возможно использование генераторов списков:

even_numbers = [num for num in numbers if num % 2 == 0]

Фильтрация позволяет выделять подмножества данных для дальнейшего анализа.

Агрегация

Агрегация – это вычисление обобщенных характеристик массива.

Основные агрегирующие операции:

- сумма элементов (sum);
- количество элементов (len);
- среднее арифметическое;
- минимальное и максимальное значение (min, max);
- подсчет количества элементов, удовлетворяющих условию.

Пример вычисления среднего значения:

average = sum(numbers) / len(numbers)

Пример подсчета количества положительных чисел:

count = 0

for num in numbers:

if num > 0:

count += 1

Комбинирование операций

На практике операции поиска, фильтрации и агрегации часто используются совместно. Например:

- отфильтровать положительные числа;
- среди них найти максимальное;
- вычислить их среднее значение.

Типичные ошибки при работе с массивами:

- отсутствие проверки на пустой массив (деление на ноль);
- неправильное условие фильтрации;
- выход за пределы индекса;
- перезапись исходных данных без необходимости;
- путаница между индексом и значением.

Практическая часть

Задание 1. Линейный поиск

1. Ввести массив из n чисел.
2. Запросить число для поиска.
3. Реализовать линейный поиск.
4. Вывести индекс найденного элемента или сообщение об отсутствии.

Задание 2. Поиск по условию

1. Найти первый элемент массива, превышающий заданное пользователем число.
2. Найти все элементы, кратные 3.
3. Вывести результаты.

Задание 3. Фильтрация массива

1. Сформировать новый массив, содержащий только положительные числа.
2. Сформировать массив чисел, лежащих в заданном диапазоне.
3. Вывести полученные массивы.

Задание 4. Агрегация данных

1. Вычислить сумму элементов массива.
2. Найти среднее арифметическое.
3. Определить количество четных и нечетных чисел.
4. Найти максимальный и минимальный элементы.

Задание 5. Комплексная обработка

1. Отфильтровать элементы, превышающие среднее значение.
2. Отсортировать полученный массив по убыванию.
3. Определить сумму элементов нового массива.

Контрольные вопросы

1. Что такое линейный поиск?
2. Чем поиск отличается от фильтрации?
3. Что такое агрегация данных?
4. Какие функции Python используются для агрегирования?
5. Почему необходимо проверять массив на пустоту перед вычислением среднего?
6. Какие ошибки возможны при обработке массивов?

ЛАБОРАТОРНАЯ РАБОТА №14.
СИМВОЛЫ И СТРОКИ. ОБРАБОТКА СТРОК

Цель работы: сформировать практические навыки работы со строковым типом данных (str) в Python, освоить основные методы и операции для обработки текстовой информации, а также научиться реализовывать типовые алгоритмы анализа и преобразования строк.

Теоретическая часть

Строки в Python (str) – упорядоченные неизменяемые последовательности символов, используемые для хранения и представления текстовой информации.

Строки – это массивы байтов, которые представляют символы Unicode. Они состоят из одного или нескольких символов, заключенных в кавычки.

Строка может заключаться как в апострофы (одинарные кавычки), так и в двойные кавычки. Строки создаются путем присваивания переменной последовательности символов, заключенных в кавычки.

Например:

S = 'spams'

S = "spams"

Строки можно писать также в тройных кавычках (когда внутри такой строки возможно присутствие кавычек и апострофов, а также можно использовать для записи многострочных блоков текста). Главное условие – внутри не должно быть трех одинаковых кавычек подряд.

Например:

S = ’’’Строка в "Python" состоит из последовательности символов.’’’

В Python нет отдельное типа для представления одного символа, символ – это просто единичная строка (S = 'a').

Строка считывается со стандартного ввода функцией input().

Строки могут выводиться на экран с использованием функции вывода.

Например:

print("привет")

Любой другой объект в Python можно перевести к строке, которая ему соответствует. Для этого нужно вызвать функцию str(),

передав ей в качестве параметра объект, переводимый в строку. Строку можно рассматривать как массив символов и соответственно обращаться к каждому символу по отдельности.

Срез (slice) – извлечение из данной строки одного символа или некоторого фрагмента подстроки или подпоследовательности.

Есть три формы срезов:

- S[i] – срез, состоящий из одного символа, который имеет номер i. При этом считается, что нумерация начинается с 0.
- S[a:b] – срез с двумя параметрами возвращает подстроку из b – a символов, начиная с символа с индексом a до символа с индексом b, не включая его.
- S[a:b:d] – срез с тремя параметрами, в котором третий параметр d задает шаг.

Примеры:

```
str = 'string'
print str[0] # Вывод первого (нулевого) символа s
print str[2] # Вывод 3-го символа (счет идет от 0)
print str[-1] # Вывод последнего символа строки g
print str[0:3] # Вывод диапазона символов от 0 до 3.
print str S[::-1] # Способ для переворота строки gnirts
print str[0:5:2] # Вывод символов от 0 до 5 с шагом в 2 символа
print str[:] # Выводим всю строку.
```

Со строками можно выполнять две арифметических операции, которые позволяют быстро преобразовывать любые последовательности символов:

1) Конкатенация (сложение)

```
s1 = 'привет '
s2 = 'мир '
print (s1 + s2) # Конкатенация строк (добавление)
привет мир
```

2) Дублирование строки (умножение)

```
print (s1*3) # Умножение строки
приветприветпривет
```

При вызове методов необходимо помнить, что строки в Python относятся к категории неизменяемых последовательностей, то есть все функции и методы могут лишь создавать новую строку. Основные функции и методы, используемыми в Python для работы со строками представлены в таблице 1.

Таблица 1 – Функции и методы строк

Метод / Операция	Назначение	Пример
str()	Преобразует объект к строковому виду	str(25) == '25'
s2 in s	Проверка, что подстрока s2 содержится в строке s	'm' in 'team'
s2 not in s	Проверка, что подстрока s2 не содержится в s	'I' not in 'team'
len(s)	Возвращает длину строки	len('abracadabra') == 11
s.find(s2) s.rfind(s2)	Возвращает индекс первого и последнего вхождения s2 в s (если нет – -1)	s = 'abracadabra' s.find('ab') == 0 s.rfind('ab') == 7 s.find('x') == -1
s.count(s2)	Количество неперекрывающихся вхождений s2 в s	'abracadabra'.count('a') == 5
s.startswith(s2) s.endswith(s2)	Проверка, начинается или заканчивается ли строка sподстрокой s2	'abracadabra'.startswith('abra')
s.join(a)	Склеивает строки из списка a, используя s как разделитель	'+'.join(['Вася', 'Маша']) == 'Вася+Маша'
s.split(s2)	Разбивает строку s по разделителю s2	'Раз два три!'.split('а') == ['Р', 'з дв', ' три!'] 'Раз два три'.split() == ['Раз', 'два', 'три']
s.replace(s2, s3)	Возвращает строку, где все неперекрывающиеся вхождения s2 заменены на s3	'Раз два три!'.replace('а', 'я') == 'Ряз двя три!'
s.isdigit() s.isalpha() s.isalnum()	Проверяет, что строка состоит только из цифр, только из букв (включая кириллицу), либо из букв и цифр	'100'.isdigit() 'abc'.isalpha() 'E315'.isalnum()
s.lower() s.upper()	Возвращает строку, где все буквы приведены к нижнему или верхнему регистру	'Привет!'.lower() == 'привет!' 'Привет!'.upper() == 'ПРИВЕТ!'
s.title()	Делает первую букву каждого слова заглавной	'привет мир!'.title() == 'Привет Мир!'
s.capitalize()	Делает заглавной только первую букву строки	'ПРИВЕТ!'.capitalize() == 'Привет!'
s.lstrip() s.rstrip() s.strip()	Удаляет пробелы (и символы пустого пространства) слева, справа или с обеих сторон	' Привет! '.strip() == 'Привет!'
s.ljust(k, c) s.rjust(k, c)	Дополняет строку символом с слева или справа до длины k	'Привет'.ljust(8, '!') == 'Привет!!'
list(s)	Преобразует строку в список символов	list('Привет') == ['П', 'р', 'и', 'в', 'е', 'т']

Практическая часть

Задание 1. Анализ строки

1. Ввести строку с клавиатуры.
2. Определить ее длину.

3. Подсчитать количество букв и цифр.
4. Определить, начинается ли строка с заглавной буквы.

Задание 2. Преобразование регистра

1. Ввести строку.
2. Вывести строку в верхнем и нижнем регистре.
3. Сделать первую букву заглавной.

Задание 3. Поиск и замена

1. Ввести текст.
2. Найти количество вхождений заданного слова.
3. Заменить это слово другим.
4. Вывести измененный текст.

Задание 4. Разбиение строки

1. Ввести предложение.
2. Разбить его на слова.
3. Вывести список слов.
4. Определить самое длинное слово.

Задание 5. Проверка строки

1. Ввести строку.
2. Проверить, является ли она палиндромом (читается одинаково слева направо и справа налево).
3. Вывести результат проверки.

Контрольные вопросы

1. Что такое строка в Python?
2. Почему строки считаются неизменяемыми?
3. Что такое срез строки?
4. Какие методы используются для поиска и замены?
5. Чем отличается split() от join()?
6. Что такое форматирование строк?
7. Какие ошибки часто возникают при работе со строками?

ЛАБОРАТОРНАЯ РАБОТА №15. ИСПОЛЬЗОВАНИЕ КОЛЛЕКЦИЙ

Цель работы: изучить основные встроенные коллекции языка Python (списки, кортежи, множества, словари), освоить операции над ними и научиться выбирать подходящую структуру данных для решения практических задач.

Теоретическая часть

Коллекция – это структура данных, предназначенная для хранения группы элементов. В Python существуют несколько основных типов коллекций:

- список (list);
- кортеж (tuple);
- множество (set);
- словарь (dict).

Каждый тип имеет свои особенности и область применения.

Список – упорядоченная изменяемая коллекция. Элементы могут изменяться, добавляться и удаляться.

numbers = [1, 2, 3]

Списки позволяют хранить элементы разных типов, но чаще используются для однотипных данных.

Кортеж – упорядоченная, но неизменяемая коллекция.

point = (3, 5)

После создания элементы кортежа изменить нельзя. Кортежи используются для хранения фиксированных данных, например координат.

Множество – неупорядоченная коллекция уникальных элементов.

nums = {1, 2, 3}

Особенности множеств:

- элементы не повторяются;
- отсутствует индексация;
- поддерживаются операции объединения, пересечения и разности.

Пример:

a = {1, 2, 3}

b = {3, 4, 5}

a | b – объединение

a & b – пересечение
a - b – разность
Словарь – коллекция пар «ключ–значение».
student = {"имя": "Иван", "оценка": 5}
Особенности словаря:

- доступ к элементу осуществляется по ключу;
- ключи должны быть уникальными;
- значения могут быть любого типа.

Основные операции:
student["имя"] – получение значения
student["курс"] = 2 – добавление новой пары
del student["оценка"] – удаление элемента
Перебор словаря:

```
for key in student:
    print(key, student[key])
```

Или:

```
for key, value in student.items():
    print(key, value)
```

Выбор подходящей коллекции:

- список – если важен порядок и возможны изменения;
- кортеж – если данные неизменяемы;
- множество – если нужны только уникальные элементы;
- словарь – если данные связаны по принципу «ключ–значение».

Практическая часть

Задание 1. Работа со списками

1. Ввести список чисел.
2. Удалить повторяющиеся элементы.
3. Отсортировать список.
4. Вывести результат.

Задание 2. Использование множества

1. Ввести два списка чисел.
2. Найти их объединение.
3. Найти пересечение.
4. Найти разность множеств.

Задание 3. Работа со словарем

1. Создать словарь, содержащий информацию о студенте (имя, возраст, курс, средний балл).
2. Добавить новое поле.

3. Изменить значение одного из ключей.
4. Вывести все пары «ключ–значение».

Задание 4. Подсчет частоты

1. Ввести строку.
2. Подсчитать частоту появления каждого символа.
3. Сохранить результат в словаре.
4. Вывести словарь частот.

Задание 5. Комплексное задание

1. Ввести список оценок студентов.
2. Используя словарь, распределить студентов по категориям:
    - «отлично» (5),
    - «хорошо» (4),
    - «удовлетворительно» (3),
    - «неудовлетворительно» (2).
3. Вывести количество студентов в каждой категории.

Контрольные вопросы

1. Какие основные коллекции существуют в Python?
2. Чем список отличается от кортежа?
3. В чем особенности множества?
4. Что такое словарь?
5. Какие требования предъявляются к ключам словаря?
6. В каких случаях целесообразно использовать множество?
7. Какие ошибки чаще всего возникают при работе с коллекциями?

ЛАБОРАТОРНАЯ РАБОТА №16.
РАБОТА С ФАЙЛАМИ. ЧТЕНИЕ И ЗАПИСЬ

Цель работы: изучить основные способы работы с файлами в языке Python, освоить операции открытия, чтения, записи и закрытия файлов, а также научиться обрабатывать текстовые данные, сохраненные во внешних источниках.

Теоретическая часть

Файл – это именованная область на носителе информации, предназначенная для хранения данных. В программировании чаще всего используются текстовые файлы (формат .txt), в которых информация представлена в виде символов.

Для работы с файлом используется функция open():

file = open("data.txt", "r")

Второй параметр определяет режим открытия:

- "r" – чтение (read);
- "w" – запись (write, создает новый файл или перезаписывает существующий);
- "a" – добавление (append, запись в конец файла);
- "r+" – чтение и запись.

После завершения работы файл необходимо закрыть:

file.close()

Рекомендуется использовать конструкцию with, которая автоматически закрывает файл:

```
with open("data.txt", "r") as file:
    content = file.read()
```

Чтение из файла

Основные способы чтения:

- read() – считывает весь файл целиком.
- readline() – считывает одну строку.
- readlines() – считывает все строки в список.

Пример построчного чтения:

```
with open("data.txt", "r") as file:
    for line in file:
        print(line.strip())
```

Метод strip() используется для удаления символа перевода строки.

Запись в файл

Для записи используется режим "w" или "a":

```
with open("output.txt", "w") as file:
    file.write("Пример строки\n")
```

Важно помнить, что режим "w" полностью очищает файл перед записью.

Запись списка строк:

```
lines = ["Первая строка\n", "Вторая строка\n"]
with open("output.txt", "w") as file:
    file.writelines(lines)
```

Кодировка

При работе с текстовыми файлами рекомендуется указывать кодировку:

```
with open("data.txt", "r", encoding="utf-8") as file:
    content = file.read()
```

Это особенно важно при работе с русским текстом.

Практическая часть

Задание 1. Чтение файла

1. Создать текстовый файл с несколькими строками.
2. Написать программу, которая:
   - открывает файл;
   - выводит его содержимое;
   - подсчитывает количество строк;
   - подсчитывает количество символов.

Задание 2. Запись в файл

1. Запросить у пользователя несколько строк текста.
2. Сохранить введенные данные в файл.
3. Проверить содержимое файла.

Задание 3. Копирование файла

1. Открыть существующий файл.
2. Считать его содержимое.
3. Записать его в новый файл.

Задание 4. Анализ текста

1. Считать текст из файла.
2. Определить количество слов.
3. Найти самое длинное слово.
4. Вывести результаты.

Задание 5. Добавление данных

1. Открыть файл в режиме добавления.
2. Добавить новую строку с текущей датой или произвольной информацией.
3. Проверить, что данные добавлены корректно.

Контрольные вопросы

1. Что такое файл?
2. Какие режимы открытия файла существуют?
3. Чем отличается read() от readline()?
4. Почему рекомендуется использовать конструкцию with?
5. В чем разница между режимами "w" и "a"?
6. Для чего указывается кодировка?
7. Какие ошибки могут возникнуть при работе с файлами?

ЛАБОРАТОРНАЯ РАБОТА №17.
РАБОТА С КАТАЛОГАМИ И ФАЙЛАМИ

Цель работы: изучить методы работы с каталогами и файлами в языке Python, освоить функции для навигации по файловой системе, проверки существования файлов и каталогов, создание и удаление, а также обработку групп файлов.

Теоретическая часть

Файл – это именованная область на носителе информации, предназначенная для хранения данных. В программировании чаще всего используются текстовые файлы (формат .txt) и бинарные файлы (например, .jpg, .bin, .csv). Текстовые файлы содержат информацию в виде символов, которые можно прочитать и изменить с помощью программ.

Каталог (папка) – это структурированный контейнер для файлов и других каталогов (подкаталогов). Каталоги позволяют организовывать данные в иерархическую структуру, облегчая поиск и обработку файлов.

Работа с каталогами и файлами важна, потому что:

- программы могут обрабатывать сразу несколько файлов;
- необходимо хранить результаты работы в разных местах;
- часто требуется автоматизация пакетной обработки данных;
- файловая система – это основной способ взаимодействия программы с внешним миром.

Навигация по каталогам

Python предоставляет несколько способов работы с каталогами, включая модуль os и объектно-ориентированный модуль pathlib.

Модуль os

- os.getcwd() – возвращает текущий рабочий каталог программы. Это важно, чтобы понимать, относительно какой папки программа будет искать или создавать файлы.
- os.chdir(path) – изменяет текущий рабочий каталог. Например, чтобы работать с файлами в другом месте.
- os.listdir(path) – возвращает список всех файлов и каталогов в указанном каталоге. Если путь не указан, используется текущий каталог.

Пример:

import os
print("Текущий каталог:", os.getcwd())
os.chdir("C:/Users/Example/Documents")
print("Содержимое каталога:", os.listdir("."))

Работа с путями

Работа с файлами требует точного указания пути к файлу:

Абсолютный путь – полный путь от корня файловой системы (C:/Users/...).

Относительный путь – путь относительно текущего рабочего каталога (data/file.txt).

Для безопасного соединения частей пути используют os.path.join():

folder = "data"
filename = "file.txt"
full_path = os.path.join(folder, filename)

Проверка существования объектов:

- os.path.exists(path) – проверка существования файла или каталога.
- os.path.isfile(path) – проверка, является ли объект файлом.
- os.path.isdir(path) – проверка, является ли объект каталогом.

Создание и удаление каталогов:

- os.mkdir(path) – создает один каталог.
- os.makedirs(path) – создает цепочку вложенных каталогов.
- os.rmdir(path) – удаляет пустой каталог.
- shutil.rmtree(path) – удаляет каталог с содержимым (для этого необходимо импортировать модуль shutil).

Пример:

import os
import shutil

os.mkdir("new_folder")
os.makedirs("parent/child")
shutil.rmtree("parent") # удаляет parent и все вложенные каталоги

Обработка файлов в каталоге

После получения списка файлов с помощью os.listdir() можно фильтровать их по типу или шаблону, например по расширению:

for file in os.listdir("."):
if file.endswith(".txt"):

print("Текстовый файл:", file)

Модуль pathlib

pathlib предоставляет объектно-ориентированный подход:

from pathlib import Path

p = Path(".")

for file in p.glob(".txt"): # находит все .txt файлы в текущем каталоге*

print(file.name)

Преимущества pathlib:

- позволяет работать с путями как с объектами, а не строками;
- упрощает создание, проверку и перебор файлов и каталогов;
- улучшает читаемость кода;
- упрощает работу с текстовыми файлами, особенно для текста на русском языке.

Практическая часть

Задание 1. Навигация и просмотр каталога

1. Определить текущий рабочий каталог.
2. Вывести список всех файлов и каталогов.
3. Перейти в другой каталог и повторить вывод.

Задание 2. Работа с файлами

1. Проверить наличие файла с заданным именем.
2. Определить, является ли объект файлом или каталогом.
3. Вывести размер файла (использовать os.path.getsize()).

Задание 3. Создание и удаление каталогов

1. Создать новый каталог.
2. Создать внутри него несколько вложенных каталогов.
3. Удалить созданные каталоги.

Задание 4. Обработка группы файлов

1. В заданном каталоге найти все текстовые файлы (.txt).
2. Прочитать каждый файл и вывести первые 5 строк.
3. Скопировать эти файлы в новый каталог (использовать shutil.copy).

Задание 5. Комплексное задание

1. В заданном каталоге найти все файлы с определенным расширением.
2. Подсчитать количество файлов каждого типа.
3. Сформировать отчет в текстовом файле с информацией: имя файла, размер, дата последней модификации.

Контрольные вопросы

1. Что такое каталог в файловой системе?
2. Чем каталог отличается от файла?
3. Как определить текущий рабочий каталог в Python?
4. Какие функции Python используются для проверки существования файлов и каталогов?
5. Как создать вложенные каталоги?
6. Как получить список всех файлов с определенным расширением?
7. Какие ошибки чаще всего возникают при работе с каталогами?

ЛАБОРАТОРНАЯ РАБОТА №18. ИСПОЛЬЗОВАНИЕ ПОДПРОГРАМ

Цель работы: изучить принципы создания и использования подпрограмм в языке Python, освоить синтаксис функций, передачу аргументов, возврат значений, область видимости переменных, а также научиться структурировать код с использованием подпрограмм.

Теоретическая часть

Подпрограмма (или функция) – это именованный блок кода, который выполняет определенную задачу и может быть вызван из разных частей программы.

Использование подпрограмм позволяет:

- уменьшить дублирование кода;
- структурировать программу;
- облегчить тестирование и отладку;
- повышать читаемость и поддерживаемость кода.

Определение функции

Функция в Python определяется с помощью ключевого слова def:

```
def имя_функции(аргументы):
    тело функции
return значение_для_возврата
```

Пример функции, вычисляющей квадрат числа:

```
def square(x):
    """Возвращает квадрат числа"""
    return x * x
print(square(5))  # вывод: 25
```

Если функция не возвращает значение, используется return без аргумента или оно опускается – в этом случае возвращается None.

Функции могут принимать:

- позиционные аргументы;
- именованные аргументы;
- аргументы со значением по умолчанию;
- переменное количество аргументов (*args и **kwargs).

Пример с аргументами по умолчанию:

```
def greet(name="Гость"):
    print(f"Привет, {name}!")
```

```
greet()          # Привет, Гость!
greet("Анна")    # Привет, Анна!
```

Область видимости переменных

Переменные делятся на:

1. **Локальные** – определены внутри функции, недоступны за ее пределами.
2. **Глобальные** – определены вне функции, могут использоваться в любой части программы.

Пример локальной переменной:

```
def func():
    x = 10  # локальная переменная
    print(x)

func()
# print(x) вызовет ошибку, так как x вне функции не существует
```

Использовать глобальные переменные внутри функции можно с ключевым словом global, но чаще рекомендуется передавать их в качестве аргументов.

Возврат нескольких значений

Python позволяет возвращать несколько значений через кортеж:

```
def calc(a, b):
    return a + b, a * b

sum_, prod = calc(3, 4)
print(sum_, prod)  # 7 12
```

Встроенные функции и модули

Python предоставляет множество встроенных функций (len(), max(), min(), sorted()) и возможность создавать собственные модули с функциями для повторного использования.

Импорт функций из другого файла:

```
from my_module import my_function
my_function()
```

Практическая часть

Задание 1. Простая функция

1. Создать функцию, принимающую число и возвращающую его квадрат.
2. Вызвать функцию с разными числами и вывести результаты.

Задание 2. Функция с несколькими аргументами

1. Создать функцию, принимающую два числа и возвращающую их сумму, разность и произведение.
2. Вызвать функцию и вывести все результаты.

Задание 3. Функция с аргументами по умолчанию

1. Создать функцию, которая приветствует пользователя, принимая имя со значением по умолчанию «Гость».
2. Проверить работу с передачей аргумента и без него.

Задание 4. Обработка списка с функцией

1. Создать функцию, которая принимает список чисел и возвращает среднее значение, минимум и максимум.
2. Вызвать функцию с разными списками и вывести результаты.

Задание 5. Использование функции для работы с файлами

1. Создать функцию для чтения файла и подсчета количества строк.
2. Создать функцию для записи списка строк в новый файл.
3. Проверить работу функций на нескольких файлах.

Контрольные вопросы

1. Что такое функция и зачем она нужна?
2. Как определить функцию в Python?
3. Чем отличаются локальные и глобальные переменные?
4. Как передавать аргументы по умолчанию?
5. Как возвращать несколько значений из функции?
6. Какие ошибки часто возникают при работе с функциями?
7. В чем преимущество использования подпрограмм при создании больших программ?

ЛАБОРАТОРНАЯ РАБОТА №19.
СЛОЖНЫЕ ФУНКЦИИ И РАЗЛИЧНЫЕ ТИПЫ ПАРАМЕТРОВ

Цель работы: изучить сложные функции в языке Python, освоить различные типы параметров (позиционные, именованные, со значением по умолчанию, переменной длины), научиться использовать вложенные функции и рекурсию, а также формировать гибкие и повторно используемые подпрограммы.

Теоретическая часть

Функции в Python могут принимать разные типы параметров:

- **Позиционные параметры** – передаются в том порядке, в котором указаны при определении функции.
- **Именованные (ключевые) параметры** – передаются с указанием имени аргумента, порядок не важен.
- **Параметры со значением по умолчанию** – используются, если аргумент не передан.
- **Произвольное количество аргументов** – *args для позиционных, **kwargs для именованных.

Пример:

```
def func(a, b=10, *args, **kwargs):
    print("a =", a)
    print("b =", b)
    print("args =", args)
    print("kwargs =", kwargs)

func(1, 2, 3, 4, x=100, y=200)
```

Результат:

```
a = 1
b = 2
args = (3, 4)
kwargs = {'x': 100, 'y': 200}
```

Вложенные функции

Функции могут быть определены внутри других функций. Внутренняя функция видна только внутри внешней и может использовать локальные переменные внешней функции.

Пример:

```
def outer(x):
    def inner(y):
```

```
            return y * 2
        return inner(x) + 1
print(outer(5))  # 11
```

Преимущества:

1. скрытие вспомогательных функций;
2. структурирование сложных алгоритмов;
3. уменьшение глобальной области видимости.

Рекурсивные функции

Рекурсия – функция вызывает сама себя. Используется для решения задач, которые естественно разбиваются на подобные подзадачи.

Пример: вычисление факториала числа:

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)

print(factorial(5))  # 120
```

Важно: всегда должно быть условие завершения, иначе возникнет ошибка переполнения стека.

Использование аргументов разных типов

Функция может принимать числа, строки, списки, словари, кортежи и даже другие функции как аргументы.

Пример передачи функции как аргумента:

```
def apply(func, value):
    return func(value)

def square(x):
    return x * x

print(apply(square, 5))  # 25
```

Практическая часть

Задание 1. Функция с разными типами параметров

1. Создать функцию, принимающую:
   - обязательный параметр;
   - параметр со значением по умолчанию;
   - произвольное количество дополнительных позиционных и именованных аргументов.

2. Вывести все аргументы в читаемом виде.

Задание 2. Вложенные функции

1. Создать внешнюю функцию, которая принимает список чисел.
2. Внутри определить функцию, возвращающую квадрат числа.
3. Внешняя функция должна возвращать новый список, где все элементы возведены в квадрат.

Задание 3. Рекурсивная функция

1. Написать рекурсивную функцию для вычисления факториала числа.
2. Написать рекурсивную функцию для нахождения суммы элементов списка.
3. Проверить работу функций на нескольких примерах.

Задание 4. Функция, принимающая другие функции

1. Создать функцию apply_to_list, которая принимает список чисел и функцию, применяет функцию к каждому элементу и возвращает новый список.
2. Проверить работу с функциями square, lambda x: x + 1.

Задание 5. Комплексное задание

1. Создать функцию для анализа текста, которая принимает текст и:
   - функцию для подсчета слов;
   - функцию для подсчета предложений;
   - функцию для нахождения самого длинного слова.
2. Вернуть результаты всех трех анализов в виде словаря.

Контрольные вопросы

1. Какие типы параметров существуют в Python?
2. Чем отличаются позиционные и именованные аргументы?
3. Что такое *args и **kwargs?
4. Как создаются вложенные функции и зачем они нужны?
5. Что такое рекурсия, и как избежать бесконечного цикла?
6. Как передавать функцию как аргумент другой функции?
7. Какие ошибки часто возникают при работе со сложными функциями?

ЛАБОРАТОРНАЯ РАБОТА №20.
СОЗДАНИЕ МОДУЛЕЙ В PYTHON

Цель работы: изучить принципы создания и использования модулей в языке Python, освоить импорт функций и переменных, понять правила организации собственного кода в отдельные файлы, а также научиться структурировать программы для повторного использования и повышения читаемости.

Теоретическая часть

Модуль – это файл с расширением .py, который содержит функции, классы и переменные, предназначенные для повторного использования в других программах.

Использование модулей позволяет:

- структурировать проект;
- избегать дублирования кода;
- использовать проверенные и готовые блоки кода;
- создавать библиотеки функций для многократного применения.

Python имеет несколько типов модулей:

1. **Встроенные модули** – поставляются вместе с Python (например, math, random, os, sys).
2. **Сторонние модули** – устанавливаются через менеджер пакетов pip (например, numpy, pandas).
3. **Пользовательские модули** – создаются самим программистом для конкретного проекта.

Создание пользовательского модуля

Для создания модуля достаточно создать файл .py, например mymodule.py, и определить в нем функции и переменные:

```
# mymodule.py
PI = 3.14159
def square(x):
    return x * x
def greet(name="Гость"):
    print(f"Привет, {name}!")
```

Импорт модулей

Существует несколько способов использования модулей в программе:

1. **Полный импорт**

```
import mymodule
```

print(mymodule.PI)
print(mymodule.square(5))
mymodule.greet("Анна")

2. **Импорт отдельных объектов**
   from mymodule import square, greet
   print(square(6))
   greet("Иван")
3. **Импорт с псевдонимом**
   import mymodule as mm
   print(mm.square(7))
   mm.greet("Ольга")
4. **Импорт всех объектов (не рекомендуется)**
   *from mymodule import **
   print(PI)
   greet()

Особенности модулей

Любой файл Python с расширением .py может быть модулем.

При первом импорте модуль выполняется полностью, а затем его содержимое кэшируется.

Для предотвращения выполнения тестового кода при импорте используется конструкция:

if __name__ == "__main__": # код, выполняемый только при запуске модуля напрямую
greet("Тест")

Модули могут содержать функции, классы, константы и переменные.

Модуль может импортировать другие модули, включая встроенные и сторонние.

Пакет – это каталог с модулями, который содержит специальный файл __init__.py.

Пример структуры пакета:

mypackage/
__init__.py
module1.py
module2.py

Импорт из пакета:

from mypackage import module1
module1.some_function()

Пакеты позволяют организовать большие проекты, разбивая функциональность на отдельные логические блоки.

Практическая часть

Задание 1. Создание модуля

1. Создать файл mymodule.py.
2. Определить в нем константу, функцию для возведения числа в квадрат и функцию приветствия.
3. Добавить проверку if __name__ == "__main__": для тестирования функций внутри модуля.

Задание 2. Импорт модуля

1. Создать новый файл main.py.
2. Импортировать модуль mymodule разными способами: полный импорт, импорт отдельных функций, с псевдонимом.
3. Проверить работу всех функций модуля.

Задание 3. Создание пакета

1. Создать каталог mypackage.
2. Внутри создать два модуля module1.py и module2.py с функциями и переменными.
3. Создать файл __init__.py, в котором импортировать функции из модулей.
4. В файле main.py импортировать пакет и использовать функции из модулей.

Задание 4. Работа с внешними модулями

1. Установить сторонний модуль через pip (например, requests).
2. Написать программу, которая использует функции установленного модуля.

Задание 5. Комплексное задание

1. Создать пакет math_utils с модулями для:
   - работы с числами;
   - работы с массивами;
   - работы со строками.
2. Создать отдельный файл для тестирования пакета, используя функции всех модулей.

Контрольные вопросы

1. Что такое модуль в Python?
2. Какие способы импорта модулей существуют?
3. Для чего используется конструкция if __name__ == "__main__":?

4. Чем пакет отличается от модуля?
5. Как организовать структуру пакета?
6. Какие ошибки часто возникают при работе с модулями?

ЛАБОРАТОРНАЯ РАБОТА №21.
РАЗРАБОТКА РЕКУРСИВНЫХ ПОДПРОГРАММ

Цель работы: изучить принципы построения рекурсивных подпрограмм в языке Python, освоить понятие базового случая и рекурсивного шага, научиться разрабатывать алгоритмы с использованием рекурсии и анализировать их корректность.

Теоретическая часть

Рекурсия – это способ организации алгоритма, при котором функция вызывает саму себя для решения подзадачи. Основная идея рекурсии заключается в разбиении задачи на более простую версию той же задачи.

Любая корректная рекурсивная функция должна содержать два обязательных элемента:

1. Базовый случай – условие завершения рекурсии. Предотвращает бесконечные вызовы.
2. Рекурсивный шаг – вызов функции с измененными параметрами, приближающими решение к базовому случаю..

Если отсутствует базовый случай, возникает бесконечная рекурсия, приводящая к ошибке переполнения стека вызовов (RecursionError).

Структура рекурсивной функции

Общая схема:

```
def func(n):
    if условие_завершения:
        return результат
    else:
        return выражение с вызовом func(меньший_аргумент)
```

Пример 1. Факториал

Факториал числа n определяется как произведение всех натуральных чисел от 1 до n.

Математическая запись:

$n! = n * (n-1)!$

При этом базовый случай:

$0! = 1$

Реализация:

```
def factorial(n):
    if n == 0 or n == 1:
```

```
        return 1
    return n * factorial(n - 1)
```

Здесь задача сводится к вычислению факториала меньшего числа.

Пример 2. Сумма первых n чисел

Рекурсивное определение:

$S(n) = n + S(n-1)$

Базовый случай:

$S(0) = 0$

Реализация:

```
def sum_n(n):
    if n == 0:
        return 0
    return n + sum_n(n - 1)
```

Механизм работы рекурсии

Каждый вызов функции помещается в стек вызовов. Когда достигается базовый случай, стек начинает «разворачиваться», возвращая значения обратно.

Важно понимать, что рекурсия – это не магия, а особый способ организации последовательных вызовов функций.

Рекурсия и итерация

Многие задачи можно решить как рекурсивно, так и с использованием циклов.

Рекурсия удобна, когда:

- задача разбивается на подобные подзадачи;
- структура данных иерархическая (деревья, вложенные списки);
- требуется лаконичное описание алгоритма.

Однако рекурсивные решения могут быть менее эффективны по памяти из-за использования стека вызовов.

Практическая часть

Задание 1. Базовая рекурсия

1. Реализовать функцию для вычисления факториала числа.
2. Реализовать функцию для вычисления суммы первых n натуральных чисел.

Задание 2. Рекурсивная работа со строками

1. Написать функцию, которая выводит строку в обратном порядке (рекурсивно).
2. Реализовать рекурсивную проверку строки на палиндром.

Задание 3. Рекурсивная работа со списками

1. Написать функцию для нахождения суммы элементов списка.
2. Написать функцию для поиска максимального элемента списка.

Задание 4. Числовые последовательности

1. Реализовать рекурсивную функцию вычисления n-го числа последовательности Фибоначчи.
2. Сравнить время работы рекурсивного и итерационного вариантов.

Задание 5. Повышенной сложности

1. Реализовать рекурсивную функцию для подсчета количества цифр в числе.
2. Реализовать функцию для вычисления степени числа (без использования оператора **).
3. Реализовать рекурсивный алгоритм поиска элемента в отсортированном списке (бинарный поиск).

Контрольные вопросы

1. Что такое рекурсия?
2. Какие элементы обязательны в рекурсивной функции?
3. Что такое базовый случай?
4. Почему возникает переполнение стека?
5. В чем отличие рекурсии от цикла?
6. В каких задачах рекурсия наиболее эффективна?
7. Какие недостатки имеет рекурсивный подход?

ЛАБОРАТОРНАЯ РАБОТА №22.
РАБОТА С КЛАССАМИ. СОЗДАНИЕ КОНСТРУКТОРОВ

Цель работы: изучить основы объектно-ориентированного программирования в языке Python, освоить создание классов и объектов, понять назначение конструктора, научиться инициализировать атрибуты экземпляра и организовывать поведение объектов через методы класса.

Теоретическая часть

Объектно-ориентированное программирование (ООП) – это парадигма программирования, основанная на представлении программы в виде совокупности объектов.

Класс – это шаблон (модель), описывающий свойства и поведение объектов.

Объект – это конкретный экземпляр класса, созданный на его основе.

Если провести аналогию, класс – это чертеж, а объект – конкретное изделие, созданное по этому чертежу.

Создание класса

В Python класс создается с помощью ключевого слова class:

```
class Person:
    pass
```

Класс может содержать:

- атрибуты – переменные, описывающие состояние объекта;
- методы – функции, определяющие поведение объекта.

Создание объекта

Объект создается вызовом класса:

```
p1 = Person()
```

Здесь p1 – экземпляр класса Person.

Атрибуты могут задаваться динамически:

```
p1.name = "Анна"
p1.age = 20
```

Однако такой способ считается неструктурированным. Более правильный подход – использовать конструктор.

Конструктор класса

Конструктор – это специальный метод **init**, который автоматически вызывается при создании объекта. Он используется для инициализации атрибутов.

Пример:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Здесь:

- self – ссылка на текущий объект;
- name и age – параметры конструктора;
- self.name и self.age – атрибуты объекта.

Создание объекта с конструктором:

```
p1 = Person("Анна", 20)
print(p1.name)
print(p1.age)
```

Методы класса

Методы – это функции, определенные внутри класса. Они работают с атрибутами объекта через self.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        print(f"Привет, меня зовут {self.name}")
```

Вызов метода:

```
p1 = Person("Анна", 20)
p1.greet()
```

Атрибуты класса и атрибуты экземпляра

Атрибуты могут быть:

- Атрибуты экземпляра – принадлежат конкретному объекту (self.name).
- Атрибуты класса – общие для всех объектов.

Пример:

```
class Student:
    university = "КузГТУ"
    def __init__(self, name):
        self.name = name
```

Здесь university – атрибут класса.

Параметры конструктора

Конструктор может содержать:

- обязательные параметры;
- параметры со значением по умолчанию;
- различные типы данных.

Пример:

```
class Car:
    def __init__(self, brand, year=2020):
        self.brand = brand
        self.year = year
```

Практическая часть

Задание 1. Создание простого класса

1. Создать класс Book.
2. Определить атрибуты: название, автор, год издания.
3. Реализовать конструктор для инициализации атрибутов.
4. Создать несколько объектов и вывести их данные.

Задание 2. Методы класса

1. Создать класс Rectangle.
2. Конструктор должен принимать длину и ширину.
3. Реализовать метод для вычисления площади.
4. Реализовать метод для вычисления периметра.
5. Создать объект и проверить работу методов.

Задание 3. Атрибуты класса

1. Создать класс Student.
2. Добавить атрибут класса university.
3. Конструктор должен принимать имя и номер группы.
4. Реализовать метод для вывода полной информации о студенте.

Задание 4. Конструктор с параметрами по умолчанию

1. Создать класс Employee.
2. Конструктор принимает имя, должность и зарплату (по умолчанию 30000).
3. Создать объекты с указанием зарплаты и без нее.

Задание 5. Комплексное задание

1. Создать класс BankAccount.
2. Конструктор принимает номер счета и начальный баланс.
3. Реализовать методы:
    - deposit (пополнение счета);
    - withdraw (снятие средств);
    - show_balance (вывод баланса).
4. Проверить корректность работы методов.

Контрольные вопросы

1. Что такое класс и объект?
2. Для чего используется метод **init**?
3. Что такое self?
4. Чем отличаются атрибуты класса от атрибутов экземпляра?
5. Когда вызывается конструктор?
6. Какие ошибки чаще всего возникают при создании классов?
7. В чем преимущества объектно-ориентированного подхода?

ЛАБОРАТОРНАЯ РАБОТА №23.
ПРИМЕНЕНИЕ СВОЙСТВ И МЕТОДОВ

Цель работы: изучить механизм создания и использования свойств (property) и методов в классах Python, научиться управлять доступом к данным объекта, реализовывать проверку значений и организовывать корректное взаимодействие между атрибутами и поведением объекта.

Теоретическая часть

В объектно-ориентированном программировании класс объединяет данные и функции для работы с ними. Данные хранятся в атрибутах объекта, а функции, определенные внутри класса, называются методами. Методы описывают поведение объекта.

Методы класса

Метод – это функция, связанная с объектом. Первый параметр метода – self – указывает на текущий экземпляр класса. Через self осуществляется доступ к атрибутам и другим методам объекта.

Пример простого метода:

```
class Person:
    def __init__(self, name):
        self.name = name
    def greet(self):
        return f"Здравствуйте, меня зовут {self.name}"
```

Методы позволяют изменять состояние объекта, выполнять вычисления, возвращать информацию.

Инкапсуляция

Инкапсуляция – это принцип сокрытия внутреннего состояния объекта и предоставления контролируемого доступа к данным. В Python принято обозначать защищенные атрибуты символом подчеркивания (например, _age). Это соглашение, а не строгий запрет.

Свойства (property)

Свойства позволяют управлять доступом к атрибутам так, чтобы обращение к ним выглядело как к обычным переменным, но фактически выполнялся метод.

Механизм property используется для:

- проверки корректности данных;
- ограничения доступа;

- вычисления значения на основе других атрибутов;
- защиты внутренних данных от прямого изменения.

Пример использования свойства:

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self._age = age
        @property
    def age(self):
        return self._age
        @age.setter
    def age(self, value):
        if value < 0:
        raise ValueError("Возраст не может быть отрицательным")
        self._age = value
```

В данном примере при обращении к student.age вызывается метод age(), а при присвоении значения – метод age.setter.

Таким образом, свойства позволяют сочетать удобство обращения к атрибуту и контроль над его изменением.

Вычисляемые свойства

Свойство может не хранить значение напрямую, а вычислять его динамически.

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
        @property
    def area(self):
        return self.width * self.height
```

Здесь area не хранится как отдельное поле, а вычисляется при каждом обращении.

Практическая часть

Задание 1. Создать класс BankAccount с атрибутами:

- владелец счета;
- баланс (защищенный атрибут).

Реализовать:

- свойство balance для получения баланса;

- метод deposit для пополнения счета;
- метод withdraw для снятия средств с проверкой достаточности средств;
- запрет на установку отрицательного баланса через setter.

Задание 2. Создать класс Employee с атрибутами:

- имя;
- оклад;
- коэффициент премии.

Реализовать:

- свойство salary с проверкой, что значение положительное;
- вычисляемое свойство total_income, возвращающее оклад с учетом премии;
- метод изменения коэффициента премии.

Задание 3. Создать класс Circle:

- атрибут радиус (с проверкой положительности);
- свойство radius с setter;
- вычисляемое свойство area;
- вычисляемое свойство circumference.

Задание 4. Создать класс Product:

- название;
- цена;
- количество на складе.

Реализовать:

- свойство price с проверкой корректности значения;
- метод sell, уменьшающий количество товара;
- метод restock для пополнения запаса;
- свойство total_value, вычисляющее общую стоимость товара на складе.

Контрольные вопросы

1. В чем отличие метода от свойства?
2. Зачем использовать property вместо обычного публичного атрибута?
3. Что происходит при обращении к свойству с setter?
4. В каких случаях удобно использовать вычисляемые свойства?
5. Как реализовать запрет на установку некорректного значения атрибута?

ЛАБОРАТОРНАЯ РАБОТА №24.
НАСЛЕДОВАНИЕ

Цель работы: изучить механизм наследования в Python, научиться создавать производные классы, переопределять методы, использовать функцию super() и строить иерархии классов для повторного использования кода.

Теоретическая часть

Наследование – это один из ключевых механизмов объектно-ориентированного программирования. Он позволяет создавать новый класс на основе уже существующего. Новый класс называется производным (или дочерним), а исходный – базовым (или родительским).

Главная идея наследования заключается в повторном использовании кода. Если несколько объектов имеют общие свойства и поведение, их удобно описать в базовом классе, а затем расширять и уточнять в дочерних классах.

Синтаксис наследования в Python:

```
class Animal:
    def __init__(self, name):
    self.name = name

    def speak(self):
    return "Животное издает звук"
class Dog(Animal):
    pass
```

Класс Dog автоматически получает доступ к атрибутам и методам класса Animal.

Расширение поведения

Производный класс может добавлять новые методы и атрибуты.

```
class Dog(Animal):
    def fetch(self):
    return f"{self.name} приносит мяч"
```

Теперь объект Dog обладает как методами базового класса, так и собственными.

Переопределение методов

Если необходимо изменить поведение унаследованного метода, его можно переопределить в дочернем классе.

```
class Dog(Animal):
    def speak(self):
        return "Гав-гав"
```

В этом случае метод speak() базового класса будет заменен новой реализацией.

Иногда требуется сохранить часть логики родительского класса. Для этого используется функция super(), которая позволяет вызвать метод базового класса.

```
class Dog(Animal):
    def __init__(self, name, breed):
    super().__init__(name)
    self.breed = breed
```

Здесь сначала вызывается конструктор родительского класса, а затем добавляется новый атрибут.

Многоуровневое наследование

Класс может наследоваться от другого производного класса.

```
class GuideDog(Dog):
    def assist(self):
        return "Собака-поводырь помогает владельцу"
```

В этом случае GuideDog наследует поведение и от Dog, и от Animal.

Множественное наследование

Python поддерживает множественное наследование – класс может иметь несколько базовых классов.

```
class Flyer:
    def fly(self):
        return "Я летаю"
class Bird(Animal, Flyer):
    pass
```

В таком случае порядок поиска методов определяется специальным алгоритмом (MRO – Method Resolution Order).

Полиморфизм

Наследование тесно связано с полиморфизмом – возможностью использовать объекты разных классов через общий интерфейс.

```
animals = [Dog("Бобик"), Animal("Неизвестный")]
for a in animals:
    print(a.speak())
```

Каждый объект вызывает свою версию метода speak(). Это делает код гибким и расширяемым.

Практическая часть

Задание 1

1. Создать базовый класс Shape с методом area(), который возвращает 0.
2. Создать производные классы Rectangle и Circle.
3. Переопределить метод area() в каждом классе.
4. Создать список фигур и вывести их площади.

Задание 2

Создать базовый класс Employee:

- имя;
- базовая зарплата;
- метод calculate_salary().

Создать производные классы:

- Manager (добавить премию);
- Developer (добавить оплату за проект).
- Переопределить метод расчета зарплаты в каждом классе.

Задание 3

Создать базовый класс Transport:

- марка;
- скорость;
- метод move().

Создать классы Car и Plane.

Переопределить метод move().

Добавить в Plane дополнительный метод take_off().

Задание 4

Реализовать многоуровневое наследование:

- класс Person;
- класс Student(Person);
- класс GraduateStudent(Student).

Добавить собственные методы на каждом уровне.

Контрольные вопросы

1. Что такое базовый и производный класс?
2. В чем преимущество наследования?
3. Что происходит при переопределении метода?
4. Для чего используется super()?
5. Что такое множественное наследование?
6. Что означает принцип полиморфизма?

ЛАБОРАТОРНАЯ РАБОТА №25. ПОЛИМОРФИЗМ И ПЕРЕОПРЕДЕЛЕНИЕ МЕТОДОВ

Цель работы: изучить принцип полиморфизма в объектно-ориентированном программировании, научиться применять переопределение методов, понимать динамическое связывание и использовать единый интерфейс для объектов разных классов.

Теоретическая часть

Полиморфизм – один из фундаментальных принципов объектно-ориентированного программирования. Сам термин происходит от греческих слов «поли» – много и «морфе» – форма. В программировании это означает, что один и тот же интерфейс может использоваться для объектов разных типов.

Идея полиморфизма заключается в том, что разные объекты могут реагировать на одно и то же сообщение (вызов метода) по-разному. При этом код, который вызывает метод, не обязан знать конкретный тип объекта. Он работает с объектом через общий интерфейс.

Представим базовый класс:

```
class Animal:
    def speak(self):
    return "Животное издает звук"
```

Теперь создадим производные классы:

```
class Dog(Animal):
    def speak(self):
    return "Гав"
class Cat(Animal):
    def speak(self):
    return "Мяу"
```

Если создать список объектов разных классов:

```
animals = [Dog(), Cat()]
for animal in animals:
print(animal.speak())
```

Будет вызвана соответствующая версия метода speak() для каждого объекта. Это и есть проявление полиморфизма.

Динамическое связывание

В Python используется динамическое связывание (dynamic binding). Это означает, что решение о том, какой именно метод будет

вызван, принимается во время выполнения программы, а не на этапе компиляции. Интерпретатор определяет тип объекта в момент вызова метода и выбирает соответствующую реализацию.

Такой механизм делает программы гибкими и расширяемыми. Можно добавлять новые классы, не изменяя код, который с ними работает, если они реализуют одинаковый интерфейс.

Переопределение методов

Переопределение – это замена метода базового класса новой реализацией в производном классе. При этом имя метода и список параметров остаются теми же.

Важно понимать, что переопределение не удаляет метод базового класса. Он остается доступным через super().

Пример:

```
class Employee:
    def calculate_salary(self):
        return 30000

class Manager(Employee):
    def calculate_salary(self):
        base = super().calculate_salary()
        return base + 20000
```

Здесь метод calculate_salary() в классе Manager расширяет поведение родительского метода.

Полиморфизм без наследования

В Python полиморфизм возможен не только через наследование. Язык поддерживает так называемую «утку-типизацию» (duck typing). Принцип формулируется так: если объект ведет себя как утка (имеет нужные методы), значит, это утка.

Пример:

```
class FileLogger:
    def write(self, message):
        print("Запись в файл:", message)
class ConsoleLogger:
    def write(self, message):
        print("Вывод в консоль:", message)
    def log(writer):
        writer.write("Сообщение")
```

Функция log() работает с любым объектом, у которого есть метод write(). Наследование здесь не требуется.

Перегрузка методов

В классическом понимании перегрузка – это создание нескольких методов с одинаковым именем, но разными параметрами. В Python строгой перегрузки нет, однако можно реализовать похожее поведение через параметры по умолчанию или проверку аргументов.

Пример:

```
class Calculator:
    def add(self, a, b=0):
        return a + b
```

Метод может работать как с одним, так и с двумя аргументами.

Полиморфизм и расширяемость

Полиморфизм делает архитектуру программы устойчивой к изменениям. Если система работает с базовым типом, можно добавлять новые производные классы без изменения уже существующего кода. Это снижает связанность компонентов программы и упрощает сопровождение.

Практическая часть

Задание 1

1. Создать базовый класс Shape с методом area().
2. Создать классы Rectangle, Circle и Triangle.
3. Переопределить метод area() в каждом классе.
4. Создать список объектов разных фигур и вывести их площади с использованием общего цикла.

Задание 2

1. Создать базовый класс Payment с методом pay().
2. Реализовать классы CardPayment, CashPayment, OnlinePayment.
3. Каждый класс должен по-своему реализовать метод pay().
4. Создать функцию process_payment(), которая принимает объект
5. Payment и вызывает метод pay().

Задание 3

1. Создать базовый класс Transport с методом move().
2. Реализовать классы Car, Boat, Plane.
3. Показать работу полиморфизма через список объектов.

Задание 4

1. Реализовать пример «утка-типизации»:
2. Создать классы Printer и EmailSender, каждый с методом send().
3. Написать функцию notify(), которая принимает объект и вызывает метод send().

Контрольные вопросы

1. Что такое полиморфизм?
2. Чем переопределение отличается от перегрузки?
3. Что такое динамическое связывание?
4. Можно ли реализовать полиморфизм без наследования?
5. Как использование super() влияет на поведение метода?
6. Почему полиморфизм повышает расширяемость программы?

ЛАБОРАТОРНАЯ РАБОТА №26. СОЗДАНИЕ ПРОЕКТА С КОМПОНЕНТАМИ ДЛЯ РАБОТЫ С ТЕКСТОМ

Цель работы: изучить принципы проектирования программ с использованием модульного подхода, научиться создавать компоненты для работы с текстовыми данными, реализовывать функции обработки строк, разделять код на логические блоки и использовать повторно созданные компоненты в проекте.

Теоретическая часть

Модульный подход в программировании предполагает разбиение программы на отдельные независимые части – модули или компоненты, каждая из которых решает определенную задачу. Такой подход улучшает читаемость, тестируемость и повторное использование кода.

Компонент – это логически завершенный блок программы, который предоставляет определенный набор функций или классов. Например, компонент для работы с текстом может содержать функции подсчета слов, поиска подстрок, замены символов, анализа текста и т. д.

Создание проекта с компонентами

Проект в Python обычно организуется в виде пакета – папки с набором модулей. Внутри проекта выделяют отдельные модули для разных функций:

- main.py – основной модуль, управляющий логикой программы;
- text_utils.py – модуль с функциями и классами для обработки текста;
- config.py – модуль с настройками и константами.

Использование модулей и пакетов

Импортирование модулей позволяет использовать их функции и классы в других частях проекта.

Пример структуры проекта:

text_project/
├── main.py
├── text_utils.py
└── config.py

Импорт функции из модуля:

from text_utils import count_words

Функции для работы с текстом

В модуле text_utils.py можно реализовать следующие функции:

Подсчет слов:

```
def count_words(text):
    return len(text.split())
```

Поиск подстроки:

```
def find_substring(text, substring):
    return substring in text
```

Замена символов:

```
def replace_char(text, old, new):
    return text.replace(old, new)
```

Анализ текста (длина, количество символов, гласные/согласные):

```
def text_statistics(text):
    vowels = "аеёиоуыэюя"
    consonants = "бвгджзйклмнпрстфхцчшщ"
    return {
    "length": len(text),
    "vowels": sum(1 for c in text.lower() if c in vowels),
    "consonants": sum(1 for c in text.lower() if c in consonants)
  }
```

Создание классов для работы с текстом

Для расширения функциональности удобно создавать классы, инкапсулирующие данные и методы:

```
class TextProcessor:
    def __init__(self, text):
        self.text = text
    def word_count(self):
        return len(self.text.split())
    def find_word(self, word):
        return word in self.text
    def replace_word(self, old, new):
        self.text = self.text.replace(old, new)
        return self.text
```

Такой класс можно использовать многократно, создавая экземпляры для разных текстов.

Преимущества компонентного подхода:

1. Повторное использование кода;
2. Легкость тестирования отдельных компонентов;

3. Возможность разделения задач между участниками проекта;
4. Упрощение поддержки и расширения проекта;
5. Улучшение читаемости и структуры кода.

Практическая часть

Задание 1

Создать модуль text_utils.py с функциями:

- подсчет слов;
- поиск подстроки;
- замена символов;
- статистика текста.

Задание 2

Создать класс TextProcessor с методами:

- word_count();
- find_word(word);
- replace_word(old, new);
- text_statistics().

Задание 3

Создать основной модуль main.py, в котором:

- импортировать модуль text_utils и/или класс TextProcessor;
- запросить у пользователя текст для обработки;
- применить функции и методы для анализа текста;
- вывести результаты в удобном формате.

Задание 4

Расширение проекта:

- реализовать поиск самых часто встречающихся слов;
- добавить возможность подсчета предложений;
- реализовать метод для перевода текста в верхний и нижний регистр.

Контрольные вопросы

1. Что такое модуль и компонент в Python?
2. В чем преимущество использования классов для работы с текстом?
3. Как организовать проект с несколькими модулями?
4. Для чего нужны функции анализа текста?
5. Чем модульный подход удобен при масштабировании проекта?

6. Как использовать повторно созданные компоненты в других проектах?

ЛАБОРАТОРНАЯ РАБОТА №27.
КНОПОЧНЫЕ КОМПОНЕНТЫ И ОБРАБОТКА СОБЫТИЙ

Цель работы: изучить принципы создания графического интерфейса пользователя (GUI) с использованием кнопочных компонентов, научиться обрабатывать события нажатия кнопок, связывать действия с событиями и строить интерактивные приложения на Python с использованием библиотеки для GUI.

Теоретическая часть

Графический интерфейс и события

В современном программировании взаимодействие с пользователем осуществляется через графический интерфейс. GUI состоит из визуальных компонентов (виджетов), таких как кнопки, текстовые поля, метки, списки, окна. Основная идея GUI-программирования – реагирование на события, происходящие с пользователем.

Событие – это действие пользователя или системы, на которое программа может реагировать. Например, клик мыши, нажатие клавиши, изменение значения поля, закрытие окна. Обработка событий осуществляется через функции или методы, называемые обработчиками событий (event handlers).

Кнопочные компоненты

Кнопка (Button) – один из базовых виджетов GUI. Она предназначена для инициирования действия при нажатии пользователем. В Python чаще всего для создания GUI используют библиотеки:

- tkinter – встроенная библиотека для создания оконных приложений;
- PyQt, PySide – для более сложных интерфейсов;
- Kivy – для кроссплатформенных приложений.

Простейший пример кнопки в Tkinter:

```
import tkinter as tk
def say_hello():
    print("Привет!")
root = tk.Tk()
button = tk.Button(root, text="Нажми меня", command=say_hello)
button.pack()
root.mainloop()
```

Здесь:

- root – главное окно приложения;
- Button – создание кнопки с текстом и назначением функции-обработчика;
- command – параметр, который связывает кнопку с функцией;
- pack() – метод размещения виджета в окне;
- mainloop() – запуск основного цикла обработки событий.

Обработка событий

Обработчик события – функция или метод, который выполняется при возникновении события. В GUI-программах основным циклом является цикл событий, который отслеживает действия пользователя и передает их соответствующим обработчикам.

Связывание кнопки с обработчиком:

button = tk.Button(root, text="Кликни", command=handler)

Когда пользователь нажимает кнопку, вызывается функция handler().

Обработка нескольких кнопок

Каждая кнопка может иметь свой обработчик. Для улучшения структуры программы часто используют объектно-ориентированный подход:

```
class App:
def __init__(self, master):
    self.button1 = tk.Button(master, text="Кнопка 1", command=self.action1)
    self.button1.pack()
    self.button2 = tk.Button(master, text="Кнопка 2", command=self.action2)
    self.button2.pack()

def action1(self):
    print("Кнопка 1 нажата")

def action2(self):
    print("Кнопка 2 нажата")

root = tk.Tk()
app = App(root)
root.mainloop()
```

Таким образом, каждый виджет может иметь индивидуальное поведение при событии.

Параметры обработчиков

Иногда требуется передать аргументы в функцию-обработчик. В Tkinter это делается с помощью лямбда-функций:

button = tk.Button(root, text="Привет", command=lambda: say_hello("Анна"))

Компоненты и расширяемость

Использование кнопок и обработчиков событий позволяет строить интерактивные приложения: формы, калькуляторы, игры, текстовые редакторы. Правильное разделение логики и интерфейса облегчает модификацию и повторное использование компонентов.

Практическая часть

Задание 1. Создать графическое окно с одной кнопкой «Привет».

При нажатии выводить сообщение «Привет, студент!».

Задание 2. Добавить вторую кнопку «Выход», которая закрывает приложение.

Задание 3. Создать класс App, в котором реализованы две кнопки и их обработчики. Кнопки должны выводить разные сообщения в консоль или на метку в окне.

Задание 4. Реализовать калькулятор с двумя кнопками:

- «Сложение» – считывает два числа и выводит сумму;
- «Умножение» – считывает два числа и выводит произведение.

Задание 5. Расширение проекта:

- добавить несколько кнопок с разными действиями; использовать параметризацию функций через lambda;
- разместить кнопки в разных частях окна с помощью методов pack, grid или place.

Контрольные вопросы

1. Что такое событие в GUI-программировании?
2. Как связать кнопку с функцией-обработчиком?
3. Как создать класс для управления несколькими кнопками?
4. В чем преимущество использования lambda-функций для обработчиков?
5. Какие библиотеки Python позволяют создавать графический интерфейс?

6. Как правильно структурировать проект с кнопочными компонентами и событиями?

ЛАБОРАТОРНАЯ РАБОТА №28. ИСПОЛЬЗОВАНИЕ ПЕРЕКЛЮЧАТЕЛЕЙ И ЭЛЕМЕНТОВ ВЫБОРА

Цель работы: изучить принципы использования переключателей (радиокнопок) и других элементов выбора в графических интерфейсах Python, научиться обрабатывать их состояние, связывать действия с выбором пользователя и создавать интерактивные приложения с вариантами выбора.

Теоретическая часть

Элементы выбора в GUI

В графическом интерфейсе элементы выбора позволяют пользователю выбирать один или несколько вариантов из предложенного набора.

Основные типы элементов выбора:

- Радиокнопки (RadioButton) – позволяют выбрать только один вариант из группы;
- Флажки (CheckButton) – позволяют выбрать несколько независимых вариантов;
- Выпадающие списки (ComboBox, OptionMenu) – позволяют выбрать один вариант из списка;
- Списки (Listbox) – позволяют выбирать один или несколько элементов.

Переключатели

Радиокнопка – это виджет, позволяющий пользователю выбрать один вариант из группы взаимосвязанных кнопок. Все кнопки в одной группе связываются с одной переменной, обычно типа IntVar или StringVar.

Пример использования радиокнопок в Tkinter:

```
import tkinter as tk

def show_choice():
print("Вы выбрали вариант:", choice.get())

root = tk.Tk()
choice = tk.IntVar()
```

rb1 = tk.Radiobutton(root, text="Вариант 1", variable=choice, value=1, command=show_choice)

rb2 = tk.Radiobutton(root, text="Вариант 2", variable=choice, value=2, command=show_choice)

rb1.pack()

rb2.pack()

root.mainloop()

Здесь переменная choice хранит значение выбранного переключателя, а функция show_choice() вызывается при изменении состояния.

Флажки

Флажки позволяют пользователю сделать несколько независимых выборов. Каждая кнопка связана с отдельной переменной типа IntVar или BooleanVar.

Пример:

var1 = tk.IntVar()

var2 = tk.IntVar()

cb1 = tk.Checkbutton(root, text="Опция A", variable=var1)

cb2 = tk.Checkbutton(root, text="Опция B", variable=var2)

cb1.pack()

cb2.pack()

Состояние каждой кнопки можно проверить через переменные var1.get() и var2.get().

Обработка событий

Элементы выбора связываются с функцией-обработчиком через параметр command. Обработчик выполняется при изменении состояния переключателя или флажка. В функции можно проверять текущее состояние переменных и выполнять соответствующие действия.

Использование выпадающих списков и списков

Выпадающие списки позволяют компактно представить несколько вариантов и выбрать один. Списки позволяют отображать набор элементов и выбирать один или несколько вариантов. В Tkinter это реализуется через OptionMenu или Listbox.

Пример OptionMenu:

options = ["Красный", "Зеленый", "Синий"]

selected = tk.StringVar()

```
selected.set(options[0])

menu = tk.OptionMenu(root, selected, *options)
menu.pack()
```

Практическая часть

Задание 1. Создать окно с двумя радиокнопками «Мужской» и «Женский». При выборе отображать выбранный вариант в консоли или на метке.

Задание 2. Добавить флажки для выбора хобби: «Спорт», «Музыка», «Чтение».
Вывести список выбранных хобби при нажатии кнопки «Показать выбор».

Задание 3. Создать выпадающий список с вариантами предметов: «Математика», «Физика», «Информатика». При выборе отображать выбранный предмет.

Задание 4. Объединить радиокнопки, флажки и выпадающий список в одном окне.
Добавить кнопку «Сохранить», которая собирает все выбранные значения и выводит их на экран.

Задание 5. Расширение проекта:

- реализовать проверку, чтобы хотя бы один флажок был выбран;
- добавить обработку ошибок при некорректном выборе;
- применить объектно-ориентированный подход, создав класс App с методами для каждого элемента выбора.

Контрольные вопросы

1. В чем отличие радиокнопки от флажка?
2. Как связать группу радиокнопок с одной переменной?
3. Как проверить состояние флажков в Tkinter?
4. Для чего используются выпадающие списки?
5. Как объединять разные элементы выбора в одном окне приложения?
6. Какие преимущества дает использование обработчиков событий для элементов выбора?

ЛАБОРАТОРНАЯ РАБОТА №29. ОТОБРАЖЕНИЕ ТАБЛИЦ

Цель работы: изучить принципы отображения и организации табличных данных в графическом интерфейсе Python, научиться создавать таблицы с использованием встроенных виджетов, управлять их содержимым, форматированием и структурой, а также реализовывать простую интерактивность с таблицами.

Теоретическая часть

Табличное представление данных – один из наиболее удобных способов визуализации структурированной информации. Таблицы используются для отображения списков, отчетов, результатов вычислений, баз данных и других структурированных наборов данных. В графических интерфейсах Python для работы с таблицами применяют:

- tkinter с виджетом Treeview (из модуля ttk);
- PyQt/PySide с виджетом QTableWidget или QTableView;
- другие библиотеки для визуализации, например, pandas + Tkinter или Kivy.

Treeview в Tkinter

В Tkinter таблицы создаются с помощью виджета Treeview из модуля ttk. Он позволяет отображать строки и столбцы, задавать заголовки, изменять ширину колонок и управлять содержимым.

Основные параметры и методы:

- columns – список имен столбцов;
- headings – названия колонок, отображаемые в заголовке;
- insert() – добавление строки в таблицу;
- delete() – удаление строки;
- item() – получение или изменение данных строки;
- selection() – работа с выбранными строками.

Пример создания таблицы:

```
import tkinter as tk
from tkinter import ttk

root = tk.Tk()
root.title("Пример таблицы")

columns = ("ID", "Имя", "Возраст")
```

```
tree = ttk.Treeview(root, columns=columns, show="headings")

for col in columns:
    tree.heading(col, text=col)
    tree.column(col, width=100)

tree.insert("", tk.END, values=(1, "Анна", 20))
tree.insert("", tk.END, values=(2, "Боб", 22))
tree.insert("", tk.END, values=(3, "Катя", 19))

tree.pack()
root.mainloop()
```

Здесь:

- show="headings" скрывает первый пустой столбец по умолчанию;
- values задает значения для каждой строки;
- метод insert() добавляет строку в конец таблицы (tk.END).

Форматирование и стили

Для улучшения визуального восприятия можно использовать стили:

- менять цвет фона и текста строк;
- выделять чередующиеся строки разными цветами;
- настраивать ширину столбцов;
- сортировать данные по столбцам (через обработчики событий).

Обработка событий с таблицей

Таблицы могут реагировать на выбор строки, двойной клик, изменение данных.

Примеры событий:

- <<TreeviewSelect>> – выбор строки;
- <Double-1> – двойной клик мышью;
- изменение содержимого через item().

Практическая часть

Задание 1. Создать окно с таблицей из трех столбцов: «ID», «Имя», «Оценка». Добавить несколько строк с тестовыми данными.

Задание 2. Добавить возможность выбора строки и вывод информации о выбранной записи в консоль.

Задание 3. Добавить кнопку «Добавить запись», которая добавляет новую строку с данными, введенными пользователем через поля ввода.

Задание 4. Добавить кнопку «Удалить запись», которая удаляет выбранную строку из таблицы.

Задание 5. Расширение проекта:

- добавить сортировку по столбцам при клике на заголовок;
- выделять чередующиеся строки разными цветами;
- добавить подсчет средней оценки и вывод результата под таблицей.

Контрольные вопросы

1. Какие виджеты Python используются для отображения таблиц в GUI?
2. Как создать таблицу с заданными столбцами и заголовками?
3. Какие методы позволяют добавлять, изменять и удалять строки в таблице?
4. Как обработать выбор строки пользователем?
5. Какие приемы позволяют улучшить визуальное восприятие таблицы?
6. В чем преимущество использования таблиц для отображения структурированных данных?

ЛАБОРАТОРНАЯ РАБОТА №30.
КОМПОНЕНТЫ ДЛЯ ОТОБРАЖЕНИЯ ДАТ И ВРЕМЕНИ

Цель работы: изучить принципы работы с компонентами отображения даты и времени в графическом интерфейсе Python, научиться получать текущую дату и время, форматировать их, отображать на экране, а также обрабатывать пользовательский ввод даты и времени.

Теоретическая часть

Дата и время в Python

Работа с датой и временем является важной частью многих приложений, таких как календари, таймеры, расписания, отчеты и планировщики. В Python для работы с датой и временем используются стандартные модули:

- datetime – для представления даты и времени, вычислений с ними и форматирования;
- time – для работы с временными интервалами и задержками;
- calendar – для работы с календарными данными, например, месяцами и днями недели.

Пример получения текущей даты и времени:

```
from datetime import datetime

now = datetime.now()
print("Текущая дата и время:", now)
```

Форматирование даты и времени

Метод strftime() позволяет задавать формат вывода:

```
print("Дата:", now.strftime("%d-%m-%Y"))
print("Время:", now.strftime("%H:%M:%S"))
```

Здесь:

- %d – день месяца;
- %m – месяц;
- %Y – год;
- %H – часы (24-часовой формат);
- %M – минуты;
- %S – секунды.

Графические компоненты для отображения даты и времени

В GUI-приложениях часто необходимо показывать текущее время или позволять пользователю выбирать дату. В Tkinter используются следующие подходы:

- Метки (Label) – для отображения текущей даты и времени;
- Entry + кнопки – для ввода даты и времени пользователем;

Специализированные виджеты из сторонних библиотек (например, tkcalendar) для выбора даты через календарь.

Пример отображения текущего времени в Label с обновлением каждую секунду:

```
import tkinter as tk
from datetime import datetime

def update_time():
    now = datetime.now().strftime("%H:%M:%S")
    label.config(text=now)
    root.after(1000, update_time)

root = tk.Tk()
label = tk.Label(root, font=("Arial", 24))
label.pack()
update_time()
root.mainloop()
```

Здесь метод after() запускает функцию обновления каждую секунду, что позволяет динамически отображать текущее время.

Использование календаря для выбора даты

Модуль tkcalendar предоставляет виджет Calendar для выбора даты:

```
import tkinter as tk
from tkcalendar import Calendar
def show_date():
    print("Выбранная дата:", cal.get_date())

root = tk.Tk()
cal = Calendar(root, selectmode='day')
cal.pack()

btn = tk.Button(root, text="Показать дату", command=show_date)
btn.pack()
```

root.mainloop()

Такой компонент упрощает ввод даты пользователем и снижает вероятность ошибок.

Практическая часть

Задание 1. Создать окно с меткой, отображающей текущее время в формате ЧЧ:ММ:СС. Обновлять время каждую секунду.

Задание 2. Добавить метку с текущей датой в формате ДД.ММ.ГГГГ.

Задание 3. Добавить поле ввода (Entry) и кнопку для ввода произвольной даты пользователем.

При нажатии кнопки выводить введенную дату в консоль.

Задание 4. Использовать виджет Calendar из tkcalendar для выбора даты. При выборе даты отображать ее в метке и консоли.

Задание 5. Расширение проекта:

- добавить выбор времени через Spinbox или Entry;
- реализовать таймер с обратным отсчетом;
- объединить дату и время в одно поле и отображать текущий момент.

Контрольные вопросы

1. Какие модули Python используются для работы с датой и временем?
2. Как получить текущую дату и время?
3. Как форматировать дату и время для отображения пользователю?
4. Какие виджеты GUI позволяют показывать и выбирать дату и время?
5. Как обновлять отображение времени в режиме реального времени?
6. В чем преимущества использования календаря для выбора даты в приложении?

ЛАБОРАТОРНАЯ РАБОТА №31.
РАЗРАБОТКА ИНТЕРФЕЙСА ПРИЛОЖЕНИЯ

Цель работы: изучить принципы проектирования графического интерфейса приложения, научиться создавать окна с различными виджетами, располагать их в логическом порядке, связывать элементы интерфейса с обработчиками событий и строить удобные для пользователя интерактивные приложения на Python.

Теоретическая часть

Интерфейс пользователя (GUI) – это средство взаимодействия пользователя с программой. Он должен быть интуитивно понятным, функциональным и эстетически оформленным.

Основные задачи интерфейса:

- отображение информации;
- прием данных от пользователя;
- управление логикой приложения через события.

В Python для разработки GUI чаще всего используют библиотеки:

- tkinter – встроенная библиотека для создания оконных приложений;
- PyQt/PySide – для более сложных и кроссплатформенных интерфейсов;
- Kivy – для мобильных и кроссплатформенных приложений.

Структура интерфейса

Интерфейс строится из окон (windows) и виджетов (widgets). Основные виджеты:

- Label – отображение текста;
- Button – кнопка для действий;
- Entry – поле ввода;
- Text – многострочное текстовое поле;
- Radiobutton/Checkbutton – элементы выбора;
- Listbox/OptionMenu – списки и выпадающие меню;
- Treeview – таблицы и списки с колонками.

Методы размещения виджетов

Tkinter предоставляет три основных метода размещения:

1. pack() – упрощенное размещение виджетов вертикально или горизонтально;

2. grid() – табличное размещение по строкам и столбцам;
3. place() – точное позиционирование по координатам.

Выбор метода зависит от сложности интерфейса. Для более сложных окон чаще используют grid(), так как он обеспечивает более точное позиционирование и управление размерами.

Обработка событий

Каждый виджет может реагировать на события пользователя: нажатие кнопки, выбор варианта, ввод текста, двойной клик мышью и т. д. Обработка событий осуществляется через функции или методы – обработчики (event handlers).

Пример кнопки с обработчиком:

```
import tkinter as tk
def greet():
    print("Привет!")
root = tk.Tk()
button = tk.Button(root, text="Скажи Привет", command=greet)
button.pack()
root.mainloop()
```

Объектно-ориентированный подход в GUI

Для более сложных приложений удобно использовать классы, инкапсулируя логику интерфейса:

```
class App:
    def __init__(self, master):
        self.master = master
        self.master.title("Приложение")
        self.label = tk.Label(master, text="Введите имя:")
        self.label.grid(row=0, column=0)
        self.entry = tk.Entry(master)
        self.entry.grid(row=0, column=1)
        self.button = tk.Button(master, text="Привет",
    command=self.say_hello)
        self.button.grid(row=1, column=0, columnspan=2)

    def say_hello(self):
        name = self.entry.get()
        print(f"Привет, {name}!")

root = tk.Tk()
app = App(root)
```

root.mainloop()

Такой подход облегчает поддержку кода, добавление новых виджетов и управление событиями.

Принципы удобного интерфейса:

1. Логичное расположение элементов;
2. Подписи к полям ввода и кнопкам;
3. Минимизация количества действий пользователя;
4. Использование стандартных элементов управления;
5. Возможность проверки корректности введенных данных.

Практическая часть

Задание 1. Создание окна

Создать главное окно приложения с заголовком и размерами 400×300.

Задание 2. Добавление поля

Добавить поле ввода имени (Entry) и метку (Label) с текстом «Введите имя:».

Задание 3. Добавление кнопки

Добавить кнопку «Привет», которая при нажатии выводит в консоль сообщение «Привет, <имя>!».

Задание 4. Добавление радиокнопки

Добавить радиокнопки для выбора пола «Мужской»/«Женский» и вывод выбранного значения при нажатии кнопки.

Задание 5. Добавление флажка

Добавить флажки с вариантами интересов или хобби. При нажатии кнопки «Показать выбор» выводить выбранные варианты.

Задание 6. Расширение проекта

Расширьте ваш проект, добавив в него следующий функционал:

- добавить таблицу (Treeview) с ранее введенными пользователями;
- сохранять данные в таблице при каждом нажатии кнопки;
- добавить кнопку «Очистить», которая очищает все поля ввода и сбрасывает выбор элементов интерфейса.

Контрольные вопросы

1. Какие основные виджеты GUI используются для взаимодействия с пользователем?

2. В чем отличие методов pack(), grid() и place() для размещения виджетов?
3. Как связать кнопку с функцией-обработчиком?
4. Какие преимущества дает использование объектно-ориентированного подхода при создании GUI?
5. Какие принципы делают интерфейс удобным для пользователя?
6. Как объединять различные элементы интерфейса в одном окне приложения?

ЛАБОРАТОРНАЯ РАБОТА №32. ТЕСТИРОВАНИЕ ПРИЛОЖЕНИЯ

Цель работы: изучить основные методы тестирования программных приложений, научиться проверять корректность работы программы, выявлять ошибки и нестандартные ситуации, а также применять автоматизированные средства тестирования для Python-приложений.

Теоретическая часть

Тестирование – это процесс проверки работоспособности программы и соответствия ее функциональности требованиям.

Основные цели тестирования:

- обнаружение ошибок и дефектов;
- подтверждение соответствия программы требованиям;
- обеспечение надежности и стабильности работы приложения.

Виды тестирования:

Ручное тестирование – проверка программы пользователем или разработчиком без использования автоматических средств. Используется для оценки интерфейса, взаимодействия с пользователем, нестандартных сценариев.

Автоматизированное тестирование – проверка программы с помощью специальных средств и скриптов, что позволяет ускорить процесс тестирования и повторно выполнять тесты при внесении изменений. В Python для этого применяются библиотеки:

- unittest – встроенный модуль для создания и запуска тестов;
- pytest – популярный инструмент для более гибкой и удобной организации тестов;
- doctest – проверка примеров, встроенных в документацию функций.

Типы тестов

Функциональные тесты – проверка правильности работы функций и методов.

Тесты пользовательского интерфейса – проверка работы GUI, обработки событий и взаимодействия с пользователем.

Регрессионное тестирование – проверка, что новые изменения не нарушили существующую функциональность.

Тесты на граничные значения – проверка работы программы с минимальными, максимальными и некорректными данными.

Тестирование исключений – проверка обработки ошибок и исключительных ситуаций.

Пример использования unittest:

```
import unittest
def add(a, b):
    return a + b
class TestMathFunctions(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(2, 3), 5)
        self.assertEqual(add(-1, 1), 0)
        self.assertEqual(add(0, 0), 0)
        if __name__ == "__main__":
            unittest.main()
```

Здесь создается класс TestMathFunctions, наследующий unittest.TestCase, и метод test_add проверяет работу функции add с разными входными данными. Метод assertEqual сравнивает ожидаемый результат с фактическим.

Тестирование GUI

Тестирование графического интерфейса сложнее, так как требуется имитация действий пользователя.

Часто используют:

- ручное тестирование кнопок, полей ввода, списков;
- проверку корректного отображения элементов;
- проверку обработчиков событий;
- автоматизированное тестирование с помощью библиотек вроде pytest-qt, pywinauto или SikuliX.

Методы тестирования:

1. **Черный ящик** – проверка функциональности без знания внутренней структуры кода. Проверяются только входные данные и результаты.
2. **Белый ящик** – анализ и тестирование внутренней логики программы, проверка всех ветвлений и циклов.
3. **Сценарное тестирование** – создание набора действий пользователя для проверки всех функций приложения.

Практическая часть

Задание 1. Создание модуля

Создать простой модуль с функцией вычисления суммы чисел и написать для него тесты с использованием unittest.

Задание 2. Создание приложения

Создать GUI-приложение с кнопкой и полем ввода. Проверить:

- корректность работы обработчика кнопки;
- обработку некорректного ввода;
- корректное обновление меток или других элементов интерфейса.

Задание 3. Добавление тестов

Добавить тесты на граничные значения для функций вычислений: минимальные и максимальные значения, нулевые и отрицательные числа.

Задание 4. Обработка исключений

Реализовать проверку обработки исключений:

- деление на ноль;
- пустой ввод в поле;
- неверный тип данных.

Задание 5. Расширение проекта

Дополнить проект:

- создать набор тестов для всех основных функций приложения;
- выполнить регрессионное тестирование после изменения кода;
- составить отчет о прохождении тестов с указанием ошибок и их исправления.

Контрольные вопросы

1. Какие основные цели тестирования программ?
2. В чем отличие ручного тестирования от автоматизированного?
3. Какие типы тестов применяются для проверки функциональности и интерфейса?
4. Что такое тестирование черного и белого ящика?
5. Какие библиотеки Python используются для автоматизированного тестирования?
6. Почему важно тестировать обработку исключений и граничные значения?

Содержание самостоятельной работы

Цель самостоятельной работы обучающихся – сформировать у студентов профессиональные компетенции в области алгоритмизации и программирования посредством выполнения практико-ориентированных заданий.

Самостоятельная работа направлена на развитие способности студентов самостоятельно решать задачи профессиональной деятельности, формирование навыков планирования времени, а также на стимулирование стремления к профессиональному развитию и совершенствованию.

Виды самостоятельной работы обучающихся указаны в таблице 1.

Таблица 1 – Виды самостоятельной работы

№ п/п	**Вид СРС**
1	Сравнение эффективности разных алгоритмов сортировки
2	Исследование скорости работы рекурсивных и итеративных алгоритмов
3	Решение задач на линейные алгоритмы (вычисления по формулам)
4	Реализация ветвлений (пример: решение квадратного уравнения, проверка високосного года)
5	Написание программ с циклами (табуляция функций, численные методы)

Обучающиеся должны изучить интернет-ресурсы и литературу по вопросам, представленным в таблице 1.

СПИСОК ЛИТЕРАТУРЫ

1. Андрианова, А. А. Алгоритмизация и программирование. Практикум : учебное пособие для СПО / А. А. Андрианова, Л. Н. Исмагилов, Т. М. Мухтарова. – 5-е изд., стер. – Санкт-Петербург : Лань, 2026. – 240 с. – ISBN 978-5-507-54545-2 // Лань : электронно-библиотечная система. – URL: https://e.lanbook.com/book/509336 (дата обращения: 12.10.2026).
2. Маркин, А. В. Программирование на sql : учебник для СПО / Маркин А. В. – Москва : Юрайт, 2025. – 435 с. – ISBN 978-5-534-11093-7. – URL: https://urait.ru/book/programmirovanie-na-sql-566220. (дата обращения: 12.10.2026).
3. Программирование. Сборник задач : учебное пособие для СПО / В. С. Батасова, П. В. Гречкина, А. А. Горкина [и др.] ; под редакцией М. М. Маран. – 3-е изд., стер. – Санкт-Петербург : Лань, 2025. – 168 с. – ISBN 978-5-507-52518-8 // Лань : электронно-библиотечная система. – URL: https://e.lanbook.com/book/454232 (дата обращения: 12.10.2026).
4. Никифоров, С. Н. Прикладное программирование : учебное пособие для СПО / С. Н. Никифоров. – 2-е изд., стер. – Санкт-Петербург : Лань, 2021. – 124 с. – ISBN 978-5-8114-8257-3 // Лань : электронно-библиотечная система. – URL: https://e.lanbook.com/book/173804 (дата обращения: 12.10.2026).
5. Агалаков, А. А. Python – от базовых конструкций до обработки данных : учебное пособие для СПО / А. А. Агалаков, К. И. Дементьева. – Саратов : Профобразование, 2025. – 109 с. – ISBN 978-5- 4488-2492-0 // Электронный ресурс цифровой образовательной среды СПО PROFобразование : [сайт]. – URL: https://profspo.ru/books/150102 (дата обращения: 12.10.2026).
6. Борзунов, С. В. Алгебра и геометрия с примерами на Python / С. В. Борзунов, С. Д. Кургалин. – 3-е изд., стер. – Санкт-Петербург : Лань, 2022. – 444 с. – ISBN 978-5-8114-9980-9 // Лань : электронно-библиотечная система. – URL: https://e.lanbook.com/book/202154 (дата обращения: 12.10.2026).
7. Кудинов, Ю. И. Основы алгоритмизации и программирования : учебное пособие для СПО / Ю. И. Кудинов, А. Ю. Келина. – 2-е изд. – Липецк ; Саратов : Липецкий государственный технический университет : Профобразование, 2020. – 71 с. – ISBN 978-5-88247-956-4, 978-5-4488-0757-2 // Электронный

ресурс цифровой образовательной среды СПО PROFобразование : [сайт]. – URL: https://profspo.ru/books/92834 (дата обращения: 12.10.2026).

8. Чурина, Т. Г. Основы алгоритмизации и программирования : учебное пособие для СПО / Т. Г. Чурина, Т. В. Нестеренко. – Саратов ; Москва : Профобразование : Ай Пи Ар Медиа, 2020. – 214 с. – ISBN 978-5-4488-0802-9, 978-5-4497-0465-8 // Электронный ресурс цифровой образовательной среды СПО PROFобразование : [сайт]. – URL: https://profspo.ru/books/96017 (дата обращения: 12.10.2026).