Оглавление

[1 Основные подходы к разработке программного обеспечения. Понятие объектно-ориентированного программирования. 2](#_Toc185979623)

[2 Общие свойства оконных классов. События, связанные с оконными классами. 2](#_Toc185979624)

[3 Графический интерфейс пользователя. Работа с таблицами 8](#_Toc185979625)

[4 Исключения в языках программирования 13](#_Toc185979626)

[5 Наследование в языках программирования 15](#_Toc185979627)

[6 Словари (хэш-таблицы) в языках программирования 18](#_Toc185979628)

[7 Ссылки в языках программирования. 19](#_Toc185979629)

[8 Типы данных в языках программирования. Операции. Примеры. 20](#_Toc185979630)

[9 Математические библиотеки и расширения в языках программирования. 21](#_Toc185979631)

[10 Расчет сложности алгоритма. Оптимизация по быстродействию 22](#_Toc185979632)

[11 Файлы в языках программирования. Классы для работы с файлами 23](#_Toc185979633)

[12 Списки в языках программирования 24](#_Toc185979634)

[13 Множества в языках программирования 25](#_Toc185979635)

[14 Функции в языках программирования. 26](#_Toc185979636)

[15 Классы (библиотеки) для работы с датой / временем в языках программирования 27](#_Toc185979637)

[16 Компоненты для связи с базами данных 28](#_Toc185979638)

[17 Модули в языках программирования 29](#_Toc185979639)

[18 Архитектура «Клиент-сервер» для работы с базами данных. Пример 30](#_Toc185979640)

[19 Структура и назначение объектов базы данных. Команды работы с объектами 31](#_Toc185979641)

[20 Итераторы в языках программирования 32](#_Toc185979642)

[21 Строки в языках программирования 33](#_Toc185979643)

[22 Язык манипулирования данными. Операторы 34](#_Toc185979644)

[23 Рекурсивные алгоритмы. Примеры 35](#_Toc185979645)

[24 Перегрузка методов. Привести примеры реализации перегрузки операций сравнения и сложения 36](#_Toc185979646)

[25 Парадигмы программирования. Структурное программирование. 37](#_Toc185979647)

[26 Графический интерфейс пользователя. Компоненты для работы с текстом 38](#_Toc185979648)

[27 Бинарные файлы. Операции. 39](#_Toc185979649)

# **1 Основные подходы к разработке программного обеспечения. Понятие объектно-ориентированного программирования.**

***Разработка программного обеспечения (ПО)*** — это процесс создания программ, который включает в себя несколько этапов. В зависимости от того, как организован этот процесс, существуют разные подходы. Рассмотрим несколько основных моделей.

1. **Каскадная модель (Waterfall)**

Каскадная модель — это классический подход, при котором весь процесс разработки делится на несколько этапов, и каждый этап завершён, прежде чем перейти к следующему. Каждый шаг должен быть выполнен полностью перед тем, как начать следующий.

*Пример***:** Разработка банковской системы, где сначала собираются все требования, проектируется система, а затем она разрабатывается и тестируется.

1. **Итеративная модель**

Итеративная модель предполагает, что проект делится на небольшие части (итерации). В каждой итерации создаётся рабочая версия программы, которая с каждым шагом становится более полной. Такой подход помогает получать результаты быстрее и вносить изменения по мере работы.

*Пример***:** Разработка сайта для интернет-магазина, где сначала создается базовая версия, а затем добавляются новые функции, такие как фильтры или способы оплаты.

1. **Инкрементная модель**

Инкрементная модель похожа на итеративную, но отличие в том, что каждый шаг даёт не просто улучшение, а полностью работоспособную часть программы, которая добавляет новые функции. После каждого шага программа становится всё более функциональной.

*Пример***:** Разработка приложения для смартфона, где сначала создаются основные функции, а затем добавляются дополнительные, например, уведомления или чат.

***Объектно-ориентированное программирование (ООП)*** — это подход к программированию, при котором программа состоит из объектов. Каждый объект содержит данные и методы для работы с этими данными. Основной принцип ООП — это моделирование реального мира с помощью объектов, которые взаимодействуют между собой. ООП основывается на четырёх ключевых принципах:

1. **Инкапсуляция**

Инкапсуляция — это принцип, при котором данные и функции, работающие с этими данными, объединяются в одном объекте. Внешние части программы не могут напрямую изменить данные объекта, а могут работать с ними только через специальные методы.

class BankAccount:

def \_\_init\_\_(self, balance):

self.\_\_balance = balance # Приватная переменная

def deposit(self, amount):

if amount > 0:

self.\_\_balance += amount

else:

print("Сумма депозита должна быть положительной.")

def get\_balance(self):

return self.\_\_balance

account = BankAccount(1000)

account.deposit(500)

print(account.get\_balance()) # 1500

Здесь переменная \_\_balance скрыта от других частей программы, и для работы с ней нужно использовать методы deposit и get\_balance.

1. **Наследование**

Наследование позволяет создать новый класс на основе уже существующего. Новый класс будет наследовать все свойства и методы старого класса, но может также добавлять новые. Это помогает избежать дублирования кода и делает программу более гибкой.

class Vehicle:

def \_\_init\_\_(self, make, model):

self.make = make

self.model = model

def start(self):

print(f"{self.make} {self.model} запускается!")

class Car(Vehicle):

def \_\_init\_\_(self, make, model, doors):

super().\_\_init\_\_(make, model)

self.doors = doors

def honk(self):

print("Бип-бип!")

my\_car = Car("Toyota", "Corolla", 4)

my\_car.start() # Toyota Corolla запускается!

my\_car.honk() # Бип-бип!

Класс Car наследует всё от класса Vehicle, но добавляет свою особенность — метод honk для сигнала.

1. **Полиморфизм**

Полиморфизм позволяет использовать одинаковые методы для разных типов объектов, и результат работы метода зависит от типа объекта. Это помогает писать универсальные функции, которые могут работать с объектами разных классов.

class Dog:

def speak(self):

return "Гав"

class Cat:

def speak(self):

return "Мяу"

def make\_animal\_speak(animal):

print(animal.speak())

dog = Dog()

cat = Cat()

make\_animal\_speak(dog) # Гав

make\_animal\_speak(cat) # Мяу

Функция make\_animal\_speak работает как с объектами типа Dog, так и с объектами типа Cat, хотя метод speak реализован по-разному в этих классах.

1. **Абстракция**

Абстракция — это принцип, который позволяет скрывать сложные детали реализации и показывать только необходимые части. Например, можно создать общий интерфейс для различных типов объектов, не вдаваясь в детали их реализации.

from abc import ABC, abstractmethod

class Animal(ABC):

@abstractmethod

def speak(self):

pass

class Dog(Animal):

def speak(self):

return "Гав"

dog = Dog()

print(dog.speak()) # Гав

Здесь создаётся абстрактный класс Animal с абстрактным методом speak, который потом реализуют конкретные классы, такие как Dog.

# **2 Общие свойства оконных классов. События, связанные с оконными классами.**

Общие свойства оконных классов

Окна в графических интерфейсах являются основными элементами, которые предоставляют пространство для взаимодействия пользователя с приложением. В Python для создания графических интерфейсов используются различные библиотеки, например, Tkinter и PyQt. Окна в этих библиотеках имеют общие свойства, которые позволяют управлять их внешним видом и поведением.

1. **Размер и положение окна**:

В обоих фреймворках можно задавать начальный размер окна с помощью методов:

* Tkinter: geometry(), например, window.geometry('300x200') — задает размер окна 300x200 пикселей.
* PyQt: resize(), например, window.resize(300, 200) — аналогичное изменение размера окна.

1. **Заголовок окна**:

В обоих фреймворках можно установить текст, который будет отображаться в заголовке окна:

* Tkinter: window.title('Заголовок')
* PyQt: window.setWindowTitle('Заголовок')

1. **Иконка окна**:

* Tkinter: метод iconbitmap() позволяет установить иконку окна.
* PyQt: метод setWindowIcon() позволяет установить иконку, используя объект QIcon.

1. **Родительские и дочерние окна**:

Окно может быть родительским для других окон. Это позволяет создать иерархию окон:

* Tkinter: создание дочернего окна с помощью Toplevel().
* PyQt: дочерние окна могут быть созданы через QDialog или QMainWindow.

События, связанные с оконными классами

События — это действия или изменения, которые происходят в окне, на которые приложение должно отреагировать. Эти события могут быть вызваны пользователем (например, щелчок мыши, нажатие клавиш) или системой (например, изменение размера окна).

1. **События от клавиатуры**:

В обоих фреймворках можно перехватывать нажатия клавиш.

Tkinter:

* bind("<KeyPress>", handler) — обработка нажатия клавиши.
* bind("<KeyRelease>", handler) — обработка отпускания клавиши.

PyQt:

* Метод keyPressEvent(self, event) позволяет перехватить нажатие клавиш.
* Метод keyReleaseEvent(self, event) обрабатывает отпускание клавиши.

1. **События изменения окна**:

Это события, которые происходят при изменении размера или положения окна.

Tkinter:

* bind("<Configure>", handler) — срабатывает при изменении размера окна.

PyQt:

* Переопределение метода resizeEvent(self, event) позволяет обрабатывать изменение размера окна.

1. **События работы с меню**:

В обоих фреймворках можно обрабатывать действия, связанные с меню.

Tkinter: используется метод add\_command для создания пункта меню и связывания его с обработчиком.

PyQt: в классе QMenu можно добавить пункты меню с помощью addAction() и связывать их с функциями через сигналы и слоты.

**Пример с Tkinter:**

import tkinter as tk

def on\_click(event):

print("Кнопка мыши была нажата.")

def on\_resize(event):

print(f"Размер окна изменен: {event.width}x{event.height}")

root = tk.Tk()

root.title("Пример окна")

root.geometry("400x300")

root.bind("<Button-1>", on\_click) # Событие нажатия левой кнопки мыши

root.bind("<Configure>", on\_resize) # Событие изменения размера окна

root.mainloop()

**Пример с PyQt:**

import sys

from PyQt5.QtWidgets import QApplication, QWidget

class MyWindow(QWidget):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.setWindowTitle("Пример окна")

self.resize(400, 300)

def mousePressEvent(self, event):

print("Кнопка мыши была нажата.")

def resizeEvent(self, event):

print(f"Размер окна изменен: {self.width()}x{self.height()}")

app = QApplication(sys.argv)

window = MyWindow()

window.show()

sys.exit(app.exec\_())

# **3 Графический интерфейс пользователя. Работа с таблицами**

**Графический интерфейс пользователя (GUI)** — это интерфейс, который позволяет пользователям взаимодействовать с программой через графические элементы, такие как кнопки, текстовые поля, метки и другие визуальные компоненты. В Python для создания GUI можно использовать несколько библиотек, наиболее популярные из которых Tkinter и PyQt.

**Пример с использованием Tkinter:**

import tkinter as tk

def on\_button\_click():

label.config(text=entry.get()) # Меняем текст метки на то, что введено в текстовое поле

# Создаем основное окно

root = tk.Tk()

root.title("Пример GUI")

# Создаем метку

label = tk.Label(root, text="Введите что-нибудь:")

label.pack()

# Создаем текстовое поле

entry = tk.Entry(root)

entry.pack()

# Создаем кнопку

button = tk.Button(root, text="Показать текст", command=on\_button\_click)

button.pack()

# Запуск главного цикла обработки событий

root.mainloop()

**Работа с таблицами в Python**

**Таблицы** в программировании обычно представляют собой структуру данных, где данные организованы в виде строк и столбцов, напоминающих таблицы в электронных таблицах, таких как Excel. В Python для работы с таблицами часто используются следующие инструменты:

1. **Pandas** — одна из самых популярных библиотек для обработки данных в виде таблиц.
2. **csv** — стандартная библиотека для работы с CSV-файлами, которые также могут быть использованы для представления таблиц.

**Работа с таблицами с использованием Pandas:**

Pandas предоставляет два основных типа объектов для работы с данными:

* **DataFrame** — таблица, которая может содержать различные типы данных.
* **Series** — одномерный массив данных.

Пример создания DataFrame и работы с ним:

import pandas as pd

# Создаем DataFrame

data = {

'Имя': ['Алиса', 'Боб', 'Карл'],

'Возраст': [25, 30, 22],

'Город': ['Москва', 'Санкт-Петербург', 'Новосибирск']

}

df = pd.DataFrame(data)

# Выводим таблицу

print(df)

# Фильтруем строки, где возраст больше 23

filtered\_df = df[df['Возраст'] > 23]

print(filtered\_df)

В этом примере:

* Мы создаем таблицу с именами, возрастами и городами.
* Используем фильтрацию данных, чтобы отобрать людей старше 23 лет.

**Работа с CSV файлами:**

Модуль csv позволяет читать и записывать таблицы в формате CSV. Например:

import csv

# Чтение CSV файла

with open('data.csv', mode='r') as file:

reader = csv.reader(file)

for row in reader:

print(row)

# Запись в CSV файл

data = [['Имя', 'Возраст'], ['Алиса', 25], ['Боб', 30]]

with open('data\_output.csv', mode='w', newline='') as file:

writer = csv.writer(file)

writer.writerows(data)

Здесь:

* Мы считываем и выводим строки из CSV файла.
* Записываем данные в новый CSV файл.

# **4 Исключения в языках программирования**

***Исключения*** — это механизм обработки ошибок в программе. Они позволяют перехватывать ошибки, которые могут возникнуть в процессе выполнения программы, и обрабатывать их вместо того, чтобы программа завершалась с ошибкой.

В Python исключения реализованы через блоки try, except, а также могут быть дополнены блоками else и finally.

**Основные концепции исключений:**

* Блок try: в этом блоке размещается код, который может вызвать исключение.
* Блок except: здесь указывается, что делать в случае возникновения исключения. Это обработчик ошибок.
* Блок else (необязателен): если ошибок не произошло, то выполняется код в блоке else.
* Блок finally (необязателен): этот блок всегда выполняется, независимо от того, было ли исключение или нет. Используется для освобождения ресурсов, таких как закрытие файлов или соединений.

Пример с обработкой исключений:

try:

x = 10 / 0

except ZeroDivisionError:

print("Ошибка деления на ноль!")

else:

print("Ошибки не возникло!")

finally:

print("Этот блок выполняется всегда.")

Объяснение:

* В блоке try пытаемся выполнить деление на ноль, что вызывает исключение ZeroDivisionError.
* В блоке except перехватывается это исключение и выводится сообщение об ошибке.
* Блок finally всегда выполнится, независимо от того, была ли ошибка.

**Типы исключений:**

Основные типы ошибок: ZeroDivisionError, ValueError, IndexError, TypeError, FileNotFoundError и другие.

**Пользовательские исключения:** можно создавать свои собственные классы исключений, наследуя от встроенного класса Exception.

Пример пользовательского исключения:

class MyError(Exception):

pass

try:

raise MyError("Произошла моя ошибка!")

except MyError as e:

print(f"Поймана ошибка: {e}")

# **5 Наследование в языках программирования**

**Наследование** — это механизм, позволяющий создавать новый класс на основе существующего, наследуя его свойства и методы. Это один из ключевых принципов объектно-ориентированного программирования (ООП).

В Python наследование реализуется следующим образом:

1. Класс-наследник (дочерний класс) получает все атрибуты и методы от родительского класса.
2. В дочернем классе можно добавлять новые атрибуты и методы или переопределять (переопределять) методы родительского класса.

Пример наследования:

class Animal:

def \_\_init\_\_(self, name):

self.name = name

def speak(self):

raise NotImplementedError("Метод speak должен быть переопределен в дочернем классе")

class Dog(Animal):

def speak(self):

return f"{self.name} говорит: Гав!"

class Cat(Animal):

def speak(self):

return f"{self.name} говорит: Мяу!"

dog = Dog("Рекс")

cat = Cat("Мурка")

print(dog.speak()) # Рекс говорит: Гав!

print(cat.speak()) # Мурка говорит: Мяу!

**Объяснение**:

* Класс Animal является родительским и содержит метод speak, который должен быть переопределен в дочерних классах.
* Классы Dog и Cat наследуют от класса Animal и реализуют свой вариант метода speak.
* Метод speak вызывается для объектов dog и cat, и каждый объект использует свою версию метода.

**Типы наследования:**

1. **Одиночное наследование**: Один класс наследует от одного родительского класса.
2. **Множественное наследование**: Класс может наследовать от нескольких классов. В Python это возможно, и язык автоматически решает, какой метод из родительских классов вызвать, с помощью метода разрешения методов (MRO — Method Resolution Order).

Пример множественного наследования:

class A:

def method\_a(self):

print("Метод A")

class B:

def method\_b(self):

print("Метод B")

class C(A, B):

def method\_c(self):

print("Метод C")

obj = C()

obj.method\_a() # Метод A

obj.method\_b() # Метод B

obj.method\_c() # Метод C

**Объяснение**:

* Класс C наследует методы как от класса A, так и от класса B.
* При создании объекта obj класса C доступны все методы, унаследованные от классов A и B.

**Переопределение методов**: Дочерний класс может переопределить метод родительского класса, чтобы изменить его поведение.

Пример:

class Animal:

def speak(self):

return "Животное издает звук"

class Dog(Animal):

def speak(self):

return "Гав!"

dog = Dog()

print(dog.speak()) # Гав!

**Использование super()**: Ключевое слово super() позволяет вызвать метод родительского класса из дочернего. Это полезно, если нужно вызвать метод родителя при переопределении.

Пример:

class Animal:

def \_\_init\_\_(self, name):

self.name = name

class Dog(Animal):

def \_\_init\_\_(self, name, breed):

super().\_\_init\_\_(name)

self.breed = breed

dog = Dog("Рекс", "Далматин")

print(dog.name) # Рекс

print(dog.breed) # Далматин

**Объяснение**:

* В классе Dog используется super().\_\_init\_\_(name), чтобы вызвать конструктор родительского класса Animal и передать ему имя.

# **6 Словари (хэш-таблицы) в языках программирования**

Словари (хэш-таблицы) — это структура данных, которая хранит пары "ключ-значение". Представьте телефонный справочник: ключ — это имя человека, а значение — его номер телефона. Чтобы найти номер, нужно знать имя.

**Как это работает:**

1. Уникальные ключи, например, имена ("Иван", "Мария").
2. Значения, связанные с ключами, например, номера (123456, 789012).
3. Вы можете быстро получить номер Марии, зная её имя.

**В Python:**

Словарь создаётся в формате {ключ: значение}, например, {"Иван": 123, "Мария": 456}.

Вы можете:

* Узнать значение по ключу: спросить "Какой номер у Ивана?" и получить 123.
* Добавить новую пару: записать в справочник нового человека.
* Удалить пару: убрать чьи-то данные.

**Изменяемость:** Словари позволяют добавлять, изменять и удалять пары. Например, вы можете обновить номер Ивана или удалить данные Марии.

Словари полезны, когда нужно организовать данные для быстрого поиска или обработки, например, в базе данных пользователей или при работе с настройками приложения.

# **7 Ссылки в языках программирования.**

Ссылки в языках программирования — это механизм, позволяющий переменной указывать на область памяти, где хранятся данные. Вместо копирования данных переменные могут ссылаться на один и тот же объект, что экономит память и упрощает обработку сложных структур.

**Особенности ссылок:**

1. **Ссылка — это указатель на объект в памяти.** Переменная "знает", где объект хранится, и может его изменять.
2. **Могут быть общими:** Несколько переменных могут ссылаться на один объект, что позволяет изменять его из разных мест программы.
3. **Зависит от типа данных:** В Python изменяемые объекты (списки, словари) передаются по ссылке, а неизменяемые (числа, строки) — по значению.

**Пример в Python:**  
Если список передаётся в функцию, то изменения внутри функции повлияют на оригинальный объект, так как функция работает с той же ссылкой.

**Практическое применение:**

* Удобно при работе с большими структурами данных (например, массивами), чтобы не дублировать данные.
* Сложности могут возникать при неосторожной работе с изменяемыми объектами, когда изменения неожиданно затрагивают другие переменные.

**Важное различие:**  
В языках с низкоуровневым доступом к памяти (например, C++) ссылки и указатели требуют явного управления памятью, а в Python ссылки скрыты от пользователя, что упрощает работу, но требует понимания концепции для предотвращения ошибок.

# **8 Типы данных в языках программирования. Операции. Примеры.**

**Типы данных в языках программирования** — это категории, которые определяют, какие значения можно хранить и какие операции с ними можно выполнять.

**Основные типы данных:**

1. **Числовые:**

* **Целые числа:** используются для целых значений, например, 42 или -7.
* **Дробные числа:** подходят для представления чисел с десятичной частью, например, 3.14.
* **Комплексные числа:** имеют реальную и мнимую части, например, 1+2i.
* Операции: сложение, вычитание, умножение, деление, возведение в степень, сравнения.

1. **Строки:**

* Хранят текстовую информацию, например слова, предложения. Операции: соединение текста, повторение, извлечение части строки.

1. **Логические:**

* Имеют два значения: истина (True) и ложь (False).
* Операции: логические "и", "или", "не", а также сравнения значений.

1. **Последовательности:**

* **Списки:** Набор элементов, которые можно изменять. Подходят для хранения данных, например списка задач.
* **Кортежи:** Набор неизменяемых элементов. Используются, если данные не должны изменяться.
* Операции: извлечение элемента, объединение последовательностей, подсчёт длины.

1. **Словари:**

* Структуры "ключ-значение". Например, можно хранить данные о человеке, где ключи — имя и возраст, а значения — их значения.
* Операции: добавление, удаление, изменение данных.

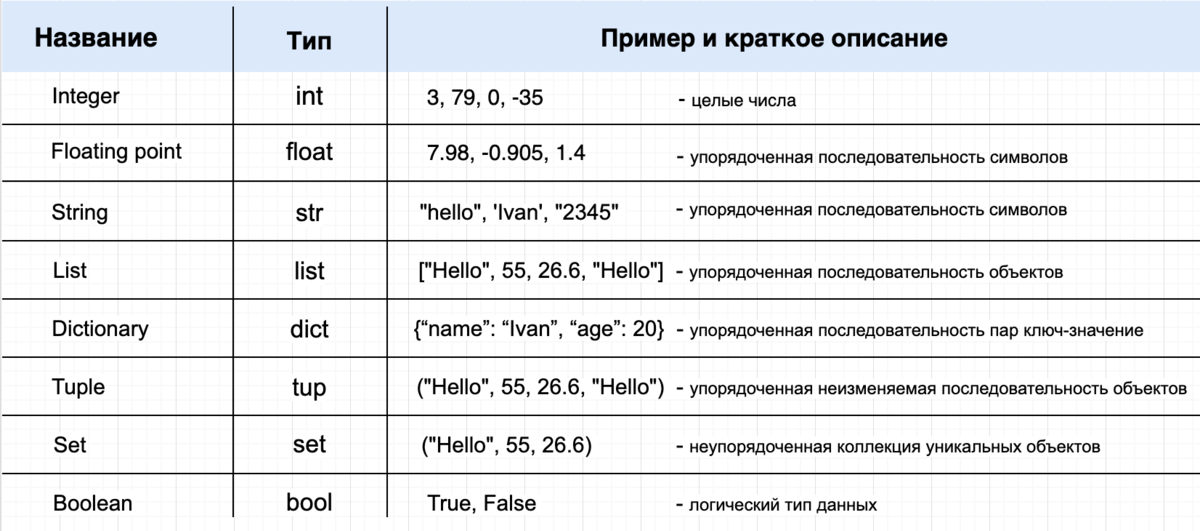
1. **Множества:**

* Хранят только уникальные элементы. Например, используются для удаления повторяющихся данных.
* Операции: объединение, пересечение, разность.

**Почему это важно:**

Тип данных определяет, как программа работает с информацией.

Например, строки удобны для работы с текстами, числовые типы — для математических операций, а словари — для структурированных данных, как в JSON.



# **9 Математические библиотеки и расширения в языках программирования.**

В языках программирования существуют различные математические библиотеки, которые предоставляют разработчикам готовые функции для выполнения стандартных и сложных математических операций. Эти библиотеки ускоряют процесс разработки, упрощают код и повышают точность вычислений.

**Библиотека math** в Python — это стандартная библиотека, которая предоставляет набор базовых математических функций. Она включает в себя операции над числами, такие как вычисление квадратного корня, логарифмов, тригонометрических функций, округление, факториалы и многие другие. Библиотека **math** особенно полезна для быстрого и эффективного выполнения базовых математических вычислений. Все функции в этой библиотеке реализованы на C, что обеспечивает хорошую производительность.

Пример использования библиотеки **math**:

* Функции для работы с числовыми значениями, например, вычисление квадратного корня или логарифмов.

Помимо **math**, существует ряд других математических библиотек, таких как **NumPy** и **SciPy**, которые предоставляют более сложные и специализированные инструменты для работы с многомерными массивами, линейной алгеброй, статистикой и численным анализом.

* **NumPy** — это библиотека для работы с массивами и матрицами, оптимизированная для вычислений с большими объемами данных.
* **SciPy** — расширение для NumPy, которое включает дополнительные функции для численных вычислений, таких как интеграция, оптимизация и обработка данных.

Таким образом, библиотеки вроде **math** подходят для стандартных задач, а более сложные библиотеки **NumPy** и **SciPy** помогают решать более специфичные задачи в научных и инженерных вычислениях.

# **10 Расчет сложности алгоритма. Оптимизация по быстродействию**

**Расчет сложности алгоритма** — это процесс оценки того, как изменяется время работы или количество используемой памяти алгоритма в зависимости от размера входных данных. Основной целью является понимание, насколько эффективно работает алгоритм, и возможно ли улучшить его производительность. Оценка сложности помогает выбрать наиболее подходящий алгоритм для решения задачи.

**Виды сложности:**

1. **Время выполнения (Time Complexity):** Оценка того, как количество операций алгоритма зависит от размера входных данных. Оценка сложности выполнения часто выражается через **асимптотику**.
   * **O(1)** — постоянное время: время выполнения не зависит от размера входных данных.
   * **O(n)** — линейное время: время выполнения пропорционально размеру входных данных.
   * **O(n²)** — квадратичное время: время выполнения пропорционально квадрату размера входных данных (например, для алгоритмов сортировки с вложенными циклами).
   * **O(log n)** — логарифмическое время: время работы растет медленно с увеличением размера данных (например, бинарный поиск).
2. **Используемая память (Space Complexity):** Оценка того, сколько памяти потребляет алгоритм в зависимости от размера входных данных.

**Пример:**

Если алгоритм работает с массивом из **n** элементов и выполняет одну операцию для каждого элемента, то его сложность по времени будет **O(n)**. Если алгоритм использует два вложенных цикла, каждый из которых проходит по всему массиву, то его сложность будет **O(n²)**.

**Оптимизация алгоритмов:**

Для повышения **быстродействия** и **эффективности** алгоритмов можно применить различные методы оптимизации:

1. **Уменьшение сложности:** Один из способов улучшить производительность — это снизить асимптотику алгоритма. Например, вместо алгоритма сортировки с квадратичной сложностью **O(n²)** можно использовать алгоритм с линейно-логарифмической сложностью **O(n log n)**, такой как **быстрая сортировка** или **сортировка слиянием**.
2. **Использование эффективных структур данных:** Некоторые структуры данных позволяют ускорить операции. Например, использование хэш-таблиц вместо простого списка может снизить сложность поиска с **O(n)** до **O(1)**.
3. **Оптимизация за счет параллелизма:** Для обработки больших объемов данных можно использовать многозадачность или многопоточность, распределяя нагрузку на несколько процессоров или ядер.
4. **Алгоритмы с жадным методом (Greedy Algorithms):** Такие алгоритмы принимают локально оптимальные решения на каждом шаге, что часто приводит к глобально оптимальному решению за меньшее количество операций.
5. **Динамическое программирование:** Этот метод позволяет сохранять результаты промежуточных вычислений, чтобы избежать повторных вычислений тех же подзадач, что существенно ускоряет выполнение алгоритма.

**Пример оптимизации:**

Предположим, есть алгоритм поиска максимального элемента в массиве. Простой способ — пройти по всему массиву и запомнить максимальное значение. Это алгоритм с **O(n)** сложностью. Однако, если данные отсортированы, можно просто взять первый элемент в отсортированном массиве, что также будет иметь сложность **O(1)**. Оптимизация здесь заключалась в том, что был использован другой подход, уменьшивший время работы.

# **11 Файлы в языках программирования. Классы для работы с файлами**

**Файлы в языках программирования** — это наборы данных, сохранённые в виде последовательности битов на носителе информации, таком как жесткий диск или SSD.

*Они являются основным средством хранения и обмена данными между программами и пользователями.*

Текстовые – те, что мы можем прочесть как человек. (.txt / .rft). Они используются для хранения конфигураций, логов и данных, которые нужно передавать между системами.

Двоичные – в файлах данные отображаются в закодированной форме 0/1. В большинстве случаев это просто последовательности битов. Они хранятся в формате .bin

Примеры двоичных файлов: .bin, .exe, .jpg, .mp3 и др.

Требует знания структуры данных – потому что файлы могут содержать сложные форматы.

**Классы и методы для работы с файлами**

В большинстве языков программирования существуют встроенные библиотеки и классы для работы с файлами. Рассмотрим несколько популярных языков:

* 1. Python:

В Python для работы с файлами используются встроенные функции open(), а также контекстный менеджер with, который автоматически управляет открытием и закрытием файлов.

with open('file.txt', 'r') as file:

content = file.read()

Для двоичных файлов используется режим 'rb' (чтение) или 'wb' (запись):

with open('image.bin', 'rb') as bin\_file:

data = bin\_file.read()

* 1. C++:

В C++ для работы с файлами используются стандартные библиотеки <fstream>, которые предоставляют классы ifstream и ofstream для чтения и записи соответственно.

Пример работы с текстовым файлом:

std::ifstream file("file.txt");

std::string line;

while (std::getline(file, line)) {

std::cout << line << std::endl;

}

# **12 Списки в языках программирования**

Списки в языках программирования — это упорядоченные наборы элементов, каждый из которых имеет свой номер, или индекс, позволяющий быстро получить к нему доступ.

В некоторых языках программирования, таких как Python, списки могут содержать элементы разных типов и изменять свой размер динамически. Это делает списки удобными для использования в ситуациях, когда требуется гибкость и возможность работы с различными типами данных.

Пример списка в Python:

mixed\_list = [1, "Hello", 3.14, True]

В этом примере список mixed\_list содержит элементы разных типов: целое число, строку, число с плавающей запятой и булево значение.

Списки называют динамическими структурами данных, потому что их можно менять на ходу: удалить один или несколько элементов, заменить или добавить новые.

# **13 Множества в языках программирования**

Множество – неупорядоченная совокупность уникальных значений хаотичного характера. Упорядоченное множество в программировании и иных науках носит название массива.

В качестве элементов множества могут выступать:

* символы;
* строки;
* числа.

Главное помнить, что перечисленные элементы должны быть константами.

В Python множества представлены встроенным типом set.

Создание множества:

my\_set = {1, 2, 3, 4}

another\_set = set([3, 4, 5, 6]) # Создание множества из списка

Основные операции:

* Добавление элемента: my\_set.add(5)
* Удаление элемента: my\_set.remove(3) (вызывает ошибку, если элемента нет) или my\_set.discard(3) (не вызывает ошибку)
* Объединение: set1.union(set2) или set1 | set2
* Пересечение: set1.intersection(set2) или set1 & set2
* Разность: set1.difference(set2) или set1 - set2

# **14 Функции в языках программирования.**

Функции в языках программирования — это блоки кода, которые выполняют определённую задачу и могут быть вызваны в любой части программы. Они принимают входные данные (параметры), выполняют действия и могут возвращать результат.

1. Определение функции: В Python функции создаются с помощью def.

def greet(name):

print(f"Hello, {name}!")

1. Параметры и аргументы: Функции могут принимать параметры, которые могут иметь значения по умолчанию.

def add(a, b=2):

return a + b

1. Возврат значения: Функции могут возвращать значение с помощью return.

def multiply(a, b):

return a \* b

1. Аргументы переменной длины: Используются \*args для неименованных аргументов и \*\*kwargs для именованных.

def sum\_all(\*args):

return sum(args)

1. Лямбда-функции: Это короткие анонимные функции, создаваемые с lambda.

multiply = lambda x, y: x \* y

1. Рекурсия: Функции могут вызывать сами себя.

def factorial(n):

if n == 0:

return 1

return n \* factorial(n - 1)

Функции позволяют организовывать код, улучшать читаемость и повторное использование. В Python функции гибкие, поддерживают аргументы переменной длины и рекурсию.

# **15 Классы (библиотеки) для работы с датой / временем в языках программирования**

В языке программирования Python для работы с датами и временем используется библиотека datetime. Она позволяет создавать, манипулировать и форматировать даты и время, а также выполнять арифметические операции с ними.

Некоторые классы библиотеки datetime:

* datetime.date. Представляет дату (год, месяц, день) без времени. Полезен для работы с календарными датами. [1](https://javarush.com/quests/lectures/ru.javarush.python.core.lecture.level08.lecture08)
* datetime.time. Представляет время (часы, минуты, секунды, микросекунды) без даты. Используется для работы с временем суток. [1](https://javarush.com/quests/lectures/ru.javarush.python.core.lecture.level08.lecture08)
* datetime.datetime. Объединяет дату и время (год, месяц, день, часы, минуты, секунды, микросекунды). Применяется для работы с конкретными моментами времени. [1](https://javarush.com/quests/lectures/ru.javarush.python.core.lecture.level08.lecture08)
* datetime.timedelta. Представляет разницу между двумя моментами времени, выраженную в днях, секундах и микросекундах. Используется для выполнения арифметических операций с датами и временем.

# **16 Компоненты для связи с базами данных**

В Python для связи с базами данных используются различные компоненты и библиотеки, которые позволяют выполнять операции по чтению, записи и обработке данных в базе данных. Основные компоненты для работы с базами данных:

**1. MySQL Connector**

Для работы с MySQL в Python используется библиотека mysql-connector-python. Она позволяет подключаться к MySQL, выполнять SQL-запросы и обрабатывать результаты.

Пример:

import mysql.connector

# Подключение к MySQL базе данных

connection = mysql.connector.connect(

host='localhost',

user='username',

password='password',

database='mydatabase'

)

cursor = connection.cursor()

# Выполнение запроса

cursor.execute('SELECT \* FROM users')

for row in cursor.fetchall():

print(row)

# Закрытие соединения

cursor.close()

connection.close()

**2. SQLite3**

SQLite3 — это встроенная библиотека в Python для работы с SQLite. SQLite является легковесной базой данных, которая хранит все данные в одном файле.

Пример:

import sqlite3

# Подключение к базе данных

connection = sqlite3.connect('example.db')

cursor = connection.cursor()

# Создание таблицы

cursor.execute('''CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY, name TEXT)''')

# Вставка данных

cursor.execute("INSERT INTO users (name) VALUES ('Alice')")

connection.commit()

# Запрос данных

cursor.execute("SELECT \* FROM users")

for row in cursor.fetchall():

print(row)

# Закрытие соединения

connection.close()

# **17 Модули в языках программирования**

**Модуль** — это файл, содержащий код, который выполняет определенную задачу. Модули позволяют организовать код, делают его более структурированным и удобным для повторного использования.

**Модули в Python**

В Python модуль — это файл с расширением .py, который может содержать функции, классы и переменные. Модули помогают разбивать программу на логические части.

**Пример модуля (math\_operations.py):**

def add(a, b):

return a + b

def multiply(a, b):

return a \* b

**Использование модуля:**

import math\_operations

result = math\_operations.add(3, 4)

**Способы импорта**

* **Полный импорт**:

import math\_operations

* **Частичный импорт**:

from math\_operations import add

* **Импорт с псевдонимом**:

import math\_operations as mo

**Стандартная библиотека Python**

Python включает модули для работы с файлами (os, sys), математикой (math), регулярными выражениями (re) и другими задачами.

**Пакеты**

Пакет — это директория с модулями и файлом \_\_init\_\_.py, который делает ее пакетом. Это помогает организовать код в более сложные структуры.

**Пример структуры пакета:**

my\_package/

\_\_init\_\_.py

module1.py

module2.py

Модули и пакеты в Python упрощают организацию и повторное использование кода.

# **18 Архитектура «Клиент-сервер» для работы с базами данных. Пример**

В архитектуре «Клиент-сервер» клиент (например, приложение) отправляет запросы на сервер, который управляет базой данных и возвращает результаты. Клиент может быть веб-приложением или десктопной программой, а сервер обрабатывает запросы к базе данных (например, MySQL).

**Как работает:**

1. **Клиент** (например, программа на Python) отправляет SQL-запрос.
2. **Сервер MySQL** обрабатывает запрос и возвращает результат клиенту.

**Пример работы с MySQL в Python:**

1. Установите библиотеку для работы с MySQL:

pip install mysql-connector-python

1. Пример кода клиента на Python:

import mysql.connector

# Подключение к серверу MySQL

connection = mysql.connector.connect(

host="localhost", # Адрес сервера MySQL

user="myuser", # Имя пользователя

password="mypassword", # Пароль

database="mydatabase" # База данных

)

cursor = connection.cursor()

# Выполнение SQL-запроса

cursor.execute("SELECT \* FROM users;")

# Получение и вывод результатов

rows = cursor.fetchall()

for row in rows:

print(row)

# Закрытие соединения

cursor.close()

connection.close()

**Как это работает:**

1. Клиент подключается к серверу MySQL, указывая параметры подключения (хост, пользователь, пароль и база данных).
2. Выполняется SQL-запрос SELECT \* FROM users;, который сервер MySQL обрабатывает и возвращает результат.
3. Сервер MySQL передает клиенту полученные данные.

**Преимущества:**

* Централизация данных на сервере.
* Масштабируемость: сервер может обслуживать множество клиентов одновременно.
* Безопасность: доступ к данным контролируется через сервер базы данных.

# **19 Структура и назначение объектов базы данных. Команды работы с объектами**

Объекты базы данных (БД) — это компоненты, с помощью которых организуются, хранятся и манипулируются данными в СУБД (системах управления базами данных). Основные объекты базы данных:

* **Таблицы** — хранят данные в виде строк и столбцов. Каждый столбец таблицы определяет тип данных, а строка — отдельную запись.
* **Представления (Views)** — виртуальные таблицы, которые выводят данные по запросу, но не хранят их физически.
* **Индексы** — структуры данных, оптимизирующие поиск и сортировку в таблицах.
* **Процедуры** — наборы SQL-операторов, которые можно вызвать для выполнения часто используемых операций.
* **Триггеры** — автоматические действия, которые выполняются при изменении данных в таблице (например, после вставки или обновления).
* **Функции** — определенные пользователем вычисления, которые могут быть использованы в запросах.

**Команды работы с объектами базы данных:**

* **Создание объектов:**
  + Для создания таблицы: CREATE TABLE  
    Пример:

CREATE TABLE students (

id INT PRIMARY KEY,

name VARCHAR(100),

age INT

);

* + Для создания представления: CREATE VIEW  
    Пример:

CREATE VIEW student\_info AS

SELECT name, age FROM students WHERE age > 18;

* + Для создания индекса: CREATE INDEX  
    Пример:

CREATE INDEX idx\_students\_name ON students(name);

* **Изменение объектов:**
  + Для изменения структуры таблицы: ALTER TABLE Пример:

ALTER TABLE students ADD COLUMN grade VARCHAR(2);

* **Удаление объектов:**
  + Для удаления таблицы: DROP TABLE Пример:

DROP TABLE students;

* **Просмотр объектов:**
  + Для просмотра информации о таблицах: SHOW TABLES или DESCRIBE  
    Пример:

SHOW TABLES;

DESCRIBE students;

# **20 Итераторы в языках программирования**

Итератор — инструмент взаимодействия, предоставляющий доступ к элементам коллекции (массива или контейнера) и навигацию по ним.  
В различных системах итераторы могут иметь разные общепринятые названия. В терминах систем управления базами данных итераторы называются курсорами. В простейшем случае итератором в низкоуровневых языках является указатель.

**Python**. Итераторы — это объекты, которые реализуют протокол итератора, позволяющий поочередно получать элементы из коллекции. Широко используются для перебора элементов последовательностей таких как списки, кортежи и строки.

**C++**. Итераторы используются чаще всего с set и list. В C++ выделяют различные типы итераторов, например, выход, вход, вперёд, двунаправленный и случайной доступа.

**C**. Итераторы позволяют управлять последовательностью элементов, обеспечивая удобный и лаконичный доступ к элементам структуры данных. С их помощью можно обрабатывать элементы коллекций без необходимости явно следить за состоянием переменных индексов.

# **21 Строки в языках программирования**

**Строковый тип** — тип данных, значениями которого является произвольная последовательность (строка) символов алфавита. Каждая переменная такого типа может быть представлена фиксированным количеством байтов либо иметь произвольную длину.  
Строка — это последовательность ASCII или UNICODE символов.

**Строки в C**, как и в большинстве языков программирования высокого уровня рассматриваются как отдельный тип, входящий в систему базовых типов языка. Так как язык C по своему происхождению является языком системного программирования, то **строковый тип данных в C как таковой отсутствует, а в качестве строк в С используются обычные массивы символов**.

**Строка в Python** — это набор символов, окруженных одинарными, двойными или тройными кавычками. Компьютер не понимает символы; внутри он хранит управляемый символ как комбинацию 0 и 1.

Каждый символ кодируется в ASCII или Unicode. Поэтому можно сказать, что строки Python также называют коллекцией символов Unicode.

В Python строки можно создавать, заключая символ или последовательность символов в кавычки.

# **22 Язык манипулирования данными. Операторы**

Язык манипулирования данными — это набор конструкций, с помощью которых можно работать с данными, их изменять, комбинировать и анализировать. В Python есть несколько типов операторов:

* **Арифметические операторы**: +, -, \*, /, //, %, \*\* (сложение, вычитание, умножение, деление и т.д.).
* **Операторы сравнения**: ==, !=, >, <, >=, <= (сравнение значений).
* **Логические операторы**: and, or, not (логические операции).
* **Операторы присваивания**: =, +=, -=, \*=, /=, и т.д.
* **Операторы битовых операций**: &, |, ^, <<, >> (работа с битами).

# **23 Рекурсивные алгоритмы. Примеры**

Рекурсия — это процесс, при котором функция вызывает сама себя для решения подзадачи. Важное условие — наличие базового случая для завершения рекурсии.

Пример: вычисление факториала числа:

def factorial(n):

if n == 0:

return 1

else:

return n \* factorial(n - 1)

Пример: Фибоначчи:

def fibonacci(n):

if n <= 1:

return n

else:

return fibonacci(n - 1) + fibonacci(n - 2)

# **24 Перегрузка методов. Привести примеры реализации перегрузки операций сравнения и сложения**

Перегрузка методов позволяет изменять поведение операторов или функций для объектов пользовательских типов.

Пример перегрузки оператора сравнения (<, ==):

class MyClass:

def \_\_init\_\_(self, value):

self.value = value

def \_\_lt\_\_(self, other):

return self.value < other.value

def \_\_eq\_\_(self, other):

return self.value == other.value

Пример перегрузки оператора сложения (+):

class Point:

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

def \_\_add\_\_(self, other):

return Point(self.x + other.x, self.y + other.y)

# **25 Парадигмы программирования. Структурное программирование.**

Парадигмы программирования — это подходы к решению задач с использованием различных стилей программирования.

**Структурное программирование** — это парадигма, которая фокусируется на организации программы через использование блоков (например, циклов, условий, функций) для обеспечения линейной, простой логики. В структурном программировании избегают использования команд перехода, таких как goto.

Пример структурного программирования:

def process\_data(data):

if not data:

return "No data"

result = []

for item in data:

result.append(item \* 2)

return result

# **26 Графический интерфейс пользователя. Компоненты для работы с текстом**

Для создания графического интерфейса пользователя в Python часто используют библиотеки, такие как **Tkinter**, **PyQt**, **wxPython**.

**Tkinter** — стандартная библиотека для создания GUI. Компоненты для работы с текстом в Tkinter:

* **Label** — для отображения текста.
* **Entry** — для ввода текста.
* **Text** — для многострочного ввода текста.

Пример с использованием Tkinter:

import tkinter as tk

root = tk.Tk()

label = tk.Label(root, text="Введите текст:")

label.pack()

entry = tk.Entry(root)

entry.pack()

root.mainloop()

# **27 Бинарные файлы. Операции.**

Бинарные файлы — это файлы, содержащие данные, представленные в двоичной форме (в отличие от текстовых файлов, которые содержат данные в виде строк текста). В бинарных файлах данные могут быть любыми: изображения, аудиофайлы, архивы, программы и другие данные, которые невозможно интерпретировать как текст.

**Операции с бинарными файлами в Python**

Для работы с бинарными файлами в Python используется встроенная функция open(), которая открывает файл в указанном режиме. Чтобы работать с бинарными файлами, нужно указать режим открытия файла с суффиксом b (например, 'rb', 'wb', 'ab' и т. д.). Операции с бинарными файлами аналогичны операциям с текстовыми, но с особенностями, связанными с обработкой данных в бинарном формате.

**Режимы открытия бинарных файлов:**

* 'rb' — чтение в бинарном режиме. Файл открывается для чтения в бинарном формате.
* 'wb' — запись в бинарном режиме. Файл открывается для записи. Если файл существует, его содержимое будет перезаписано.
* 'ab' — добавление в бинарном режиме. Открывает файл для записи, но не стирает данные, а добавляет новые данные в конец файла.
* 'rb+' — чтение и запись в бинарном режиме. Открывает файл для чтения и записи.
* 'wb+' — запись и чтение в бинарном режиме.

**Примеры операций с бинарными файлами:**

1. **Чтение бинарного файла**: Для чтения данных из бинарного файла используется метод read(), который возвращает данные в виде байтов.

# Открытие файла для чтения в бинарном режиме

with open('file.bin', 'rb') as file:

data = file.read() # Чтение всех данных

print(data) # Вывод данных (например, байтовое представление)

1. **Запись в бинарный файл**: Для записи данных в бинарный файл используется метод write(). Важно помнить, что записывать нужно байты, а не строки.

# Открытие файла для записи в бинарном режиме

with open('file.bin', 'wb') as file:

file.write(b'\x01\x02\x03') # Запись байтов

1. **Запись и чтение файлов в бинарном режиме с использованием rb+ и wb+**: Открытие файла для чтения и записи, чтобы изменить его содержимое.

# Открытие файла для чтения и записи в бинарном режиме

with open('file.bin', 'rb+') as file:

data = file.read(3) # Чтение первых 3 байтов

print(data) # Вывод прочитанных байтов

file.seek(0) # Возврат в начало файла

file.write(b'\x04\x05\x06') # Запись новых данных

1. **Запись данных в файл с добавлением новых данных**: При использовании режима 'ab' данные добавляются в конец файла.

# Открытие файла для добавления данных в бинарном режиме

with open('file.bin', 'ab') as file:

file.write(b'\x07\x08\x09') # Добавление данных

1. **Чтение определённого количества байтов**: Метод read() позволяет указывать количество байтов для чтения.

with open('file.bin', 'rb') as file:

data = file.read(5) # Чтение 5 байтов

print(data)

1. **Использование метода seek() для перемещения по файлу**: Метод seek() позволяет перемещаться по файлу и менять позицию чтения/записи.

with open('file.bin', 'rb') as file:

file.seek(5) # Перемещение на 5-й байт

data = file.read(3) # Чтение 3 байтов, начиная с 5-й позиции

print(data)

**Важно:**

Бинарные файлы могут содержать данные, которые не могут быть интерпретированы как текст. Например, символы и байты могут иметь значения, которые не отображаются в виде читаемых символов.

Работая с бинарными файлами, важно точно понимать, какие данные содержатся в файле (например, изображение, аудио или другие специфичные данные), и как они должны быть обработаны.