**Системы контроля версий.**

**Система контроля версий** **(*Version Control System, VCS*)** представляет собой программное обеспечение, которое позволяет отслеживать изменения в документах, при необходимости производить их откат, определять, кто и когда внес исправления и т.п.

Системы контроля версий можно разделить на две группы: распределенные и централизованные. **Централизованные** системы контроля версий представляют собой приложения типа клиент-сервер, когда репозиторий проекта существует в единственном экземпляре и хранится на сервере. **Распределенные** системы контроля версий (*Distributed Version Control System, DVCS*) позволяют хранить репозиторий (его копию) у каждого разработчика, работающего с данной системой. *Git* – распределенная система контроля версий,

**Ключевые слова, идентификаторы**

**Ключевые слова** — это часть языка, зарезервированные слова в Python. Ключевое слово нельзя использовать в качестве имени переменной, функции или любого другого идентификатора. Они нужны для синтаксиса и структуры языка Python.

Ключевые слова в Python чувствительны к регистру. Все, кроме True, False и None, пишутся в нижнем регистре.

**Идентификатор** — имя некоторого объекта в программе, являющееся его уникальным признаком, позволяющим отличать его от других объектов.

Идентификатор обязательно есть у каждой переменной, функции, объекта и т.п. Идентификаторы в Питоне не ограничены по длине и чувствительны к регистру.

В идентификаторах допустимы только символы от "A" до "Z" в верхнем и нижнем регистре, подчеркивание "\_" и, кроме первого символа идентификатора, цифры от "0" до "9".

**Правила записи идентификаторов**

1. В записи идентификаторов можно использовать латинские буквы в нижнем регистре (от a до z), в верхнем регистре (от A до Z), цифры (от 0 до 9) и символ подчеркивания \_. Например, допустимыми будут такие идентификаторы: myClass, var\_1 и print\_this\_to\_screen.

2. Идентификатор не может начинаться с цифры.

3. Ключевые слова нельзя использовать в качестве идентификаторов.

global = 1

4. При записи идентификатора нельзя использовать специальные символы, например: ! , @ , # , $ , % и т. д.

5. Идентификатор может быть любой длины.

**Фундаментальные типы данных. Переменные и константы.**

Python поддерживает **динамическую типизацию**: переменная получает тип данных в момент присваивания.

**Фундаментальные типы данных:**

* int — целое число: 5, -10
* float — число с плавающей точкой: 3.14, -0.01
* bool — логический тип: True, False
* str — строка: "привет"
* None — специальный тип для "пустых" значений

**Переменные** — хранят данные. Объявляются без указания типа.

Для определения типа переменных есть функция type.

Некоторые данные никогда не меняются — например, математические постоянные. Возьмем для примера число π. Оно всегда равно 3.14 и не может измениться. Чтобы обратиться к подобным данным, в Python используют константы. Константа создается так же, как переменная. Разница только в том, что константы принято именовать заглавными буквами и с \_ в качестве разделителя между словами. Константа, как и переменная, может использоваться в любом выражении. По своим свойствам константа не отличается от переменной и может быть изменена.

**Основные управляющие структуры и операторы**

**Условные операторы**

Используются для выполнения разных действий в зависимости от условий.

**if-elif-else**

**Циклы**

Позволяют выполнять блок кода несколько раз.

**for** – перебор элементов последовательности (списка, строки, диапазона).

**while** – выполняется, пока условие истинно.

**Операторы**

**Арифметические:**

* + (сложение), - (вычитание), \* (умножение), / (деление), % (остаток), \*\* (степень).

**Сравнения:**

* == (равно), != (не равно), > (больше), < (меньше), >= (больше или равно), <= (меньше или равно).

**Логические:**

* and (И), or (ИЛИ), not (НЕ).

**Присваивания:**

* = (присвоение), += (увеличение на значение), -= (уменьшение на значение).

**Операторы управления циклом**

* break – досрочный выход из цикла.

**Приведение и преобразование типов, присваивание**

Данные, которые соответствуют переменной в языке Python, могут быть определены с помощью инструкции присваивания. Инструкция присваивания в языке Python записывается с помощью знака равенства, слева от которого находится имя переменной, а справа – её значение. То есть, чтобы переменной а присвоить значение 4, мы должны записать строку кода: а = 4.

**Особенности**: Не все преобразования возможны (например, строку с буквами нельзя преобразовать в число)

При арифметических операциях Python автоматически преобразует типы, когда это необходимо (например, int + float → float)

|  |  |
| --- | --- |
| Метод | Что делает |
| list.append(x) | Добавляет элемент в конец списка |
| list.extend(L) | Расширяет список list, добавляя в конец все элементы списка L |
| list.insert(i, x) | Вставляет на i-ый элемент значение x |
| list.remove(x) | Удаляет первый элемент в списке, имеющий значение x. ValueError, если такого элемента не существует |
| list.pop([i]) | Удаляет i-ый элемент и возвращает его. Если индекс не указан, удаляется последний элемент |
| list.index(x, [start [, end]]) | Возвращает положение первого элемента со значением x (при этом поиск ведется от start до end) |
| list.count(x) | Возвращает количество элементов со значением x |
| list.sort([key=функция]) | Сортирует список на основе функции |
| list.reverse() | Разворачивает список |
| list.copy() | Поверхностная копия списка |
| list.clear() | Очищает список |

**Операторы выбора**

**if else** — это оператор, управляющий условным ветвлением. Простыми словами это конструкция в Python, указывающая интерпретатору, следует ли выполнять определенный участок кода или нет.

Как и все прочие составные инструкции языка, оператор выбора также поддерживает свойство вложенности. Это означает, что использование if else позволяет создавать внутри программного модуля так называемое логическое ветвление.

**Синтаксис**

Оператор if else в языке Python — это типичная условная конструкция, которую можно встретить и в большинстве других языков программирования.

Синтаксически конструкция выглядит следующим образом:

1. сначала записывается часть if с условным выражением, которое возвращает истину или ложь;
2. затем может следовать одна или несколько необязательных частей elif (в других языках вы могли встречать else if);
3. Завершается же запись этого составного оператора также необязательной частью else.

## **Оператор elif**

elif позволяет программе выбирать из нескольких вариантов. Это удобно, например, в том случае, если одну переменную необходимо многократно сравнить с разными величинами.

**Заглушка pass**

Оператор-заглушка pass заменяет собой отсутствие какой-либо операции.

**Конструкция switch case**

В python данная конструкция отсутствует, однако начиная с версии 3.10 была введена аналогичная конструкция – match/case.

Оператор match был введен для того, чтобы быть больше чем просто похожим на оператор switch, который присутствует во многих других языках программирования.

Во многих случаях конструкция match/case, может упростить и повысить читабельность кода Python.

**Операторы циклов**

**Циклы в Python** – инструменты, упрощающие работу с кодом. В программировании все циклично и последовательно. Написание любой программы состоит из прохождения определенных этапов, задачи в которых могут повторяться.

В Python основных циклов всего два – «while» и «for». Первый используется тогда, когда заранее известно количество итераций, а второй – когда нужно выполнить перебор элементов.

* Единоразовое выполнение — так называемая итерация.
* Тело цикла — последовательность кода, которую можно исполнить многократно.

Применительно к Python можно выделить следующие плюсы использования циклов:

* Возможность перебора элементов структур данных (массивов или связанных списков).
* Возможность многократного использования кода.
* Отсутствие необходимости прописывать один и то же код несколько раз.

**Выход из цикла Python или его смена**

Чтобы произвести выход из цикла в программировании на Python применяется оператор «break». Он осуществляет досрочное завершение, обходя «else».

Ещё одна инструкция, вносящая изменения в цикл, — «continue». При написании данного оператора внутри кода программа игнорирует все остальные инструкции до конца цикла. Далее начинается следующая итерация.

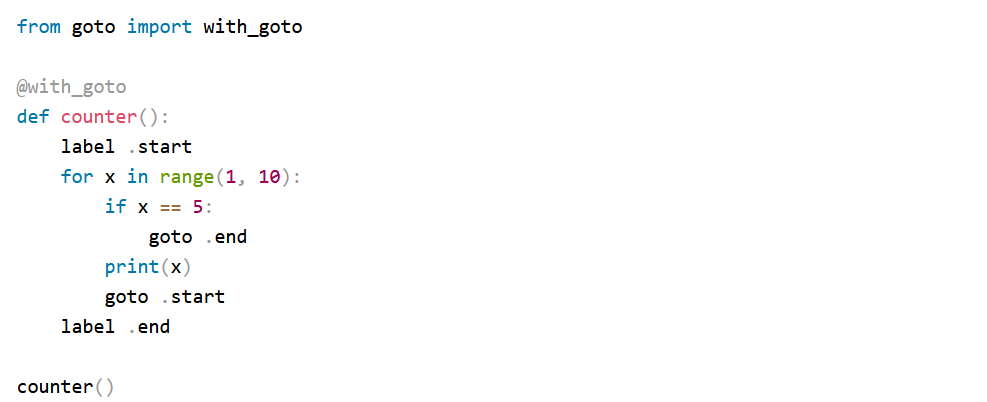
**Бесконечные циклы**

В этом случае не выполняется условие выхода. Скажем, цикл «while» является таковым в том случае, если его условие не может быть ложным. Он полезен, к примеру, для создания программы «Часы», беспрерывно демонстрирующей время.

**Операторы перехода**

Изучая Python, вы заметите, что здесь нет места для goto и меток. Python — это язык с чётко выстроенной структурой и высокой читабельностью, где не приемлем хаос. Рассмотрим альтернативы, предоставляемые Python.

Прямого аналога команде goto в Python нет. Но существуют приёмы, эмулирующие её поведение. Вот пример имитации goto с помощью сторонней библиотеки:



Хоть данный код и работает, рекомендуется придерживаться структурированного сценария управления потоком в Python вместо применения goto. Именно к этому стремится Python 🐍.

**Алгоритмы обработки числовых данных**

В Python обработка чисел может включать:

* арифметические операции (+, -, \*, /, \*\*, %);
* округление (round());
* возведение в степень (pow() или \*\*);
* использование модулей math, decimal, fractions для работы с точными числами.

**Коллекции (комплексные типы данных)**

В языке Python существует несколько комплексных типов данных, каждый из которых имеет свои отличия и особенности: set, list, tuple, dictionary, frozenset.

## **List.**

Списки в Python - упорядоченные изменяемые коллекции объектов произвольных типов (почти как массив, но типы могут отличаться).

Чтобы использовать списки, их нужно создать. Создать список можно несколькими способами. Например, можно обработать любой итерируемый объект (например, строку) встроенной функцией list

И еще один способ создать список - это генераторы списков. Генератор списков - способ построить новый список, применяя выражение к каждому элементу последовательности. Генераторы списков очень похожи на цикл for.

## **Tuple.**

Кортеж, по сути - неизменяемый [список](https://pythonworld.ru/tipy-dannyx-v-python/spiski-list-funkcii-i-metody-spiskov.html).

## **Dictionary.**

Словари в Python - неупорядоченные коллекции произвольных объектов с доступом по ключу. Их иногда ещё называют ассоциативными массивами или хеш-таблицами.

Чтобы работать со словарём, его нужно создать. Сделать это можно несколькими способами. Во-первых, с помощью литерала.

Во-вторых, с помощью функции dict.

В-третьих, с помощью метода fromkeys:

В-четвертых, с помощью генераторов словарей, которые очень похожи на [генераторы списков](https://pythonworld.ru/tipy-dannyx-v-python/spiski-list-funkcii-i-metody-spiskov.html).

## **Set и frozenset.**

Множество в python (set) – "контейнер", содержащий не повторяющиеся элементы в случайном порядке.

**Библиотека numpy. Функционал и возможности.**

NumPy — это библиотека языка Python, добавляющая поддержку больших многомерных массивов и матриц, вместе с большой библиотекой высокоуровневых (и очень быстрых) математических функций для операций с этими массивами.

Основным объектом NumPy является однородный многомерный массив (в numpy называется numpy.ndarray). Это многомерный массив элементов (обычно чисел), одного типа.

Математические операции над массивами выполняются поэлементно. Создается новый массив, который заполняется результатами действия оператора.

**Библиотека pandas. Функционал и возможности.**

**Pandas** – это высокоуровневая [Python](https://khashtamov.com/category/python/) библиотека для анализа данных. Почему я её называю высокоуровневой, потому что построена она поверх более низкоуровневой библиотеки NumPy (написана на Си), что является большим плюсом в производительности. В экосистеме [Python](https://khashtamov.com/2016/06/why-python/), pandas является наиболее продвинутой и быстроразвивающейся библиотекой для обработки и анализа данных.

**Чтение и запись данных**

pandas поддерживает все самые популярные форматы хранения данных: csv, excel, sql, буфер обмена, html и многое другое:

Считать данные из csv-файла и превратить в DataFrame можно функцией read\_csv.

**Группировка и агрегирование в pandas**

Группировка данных один из самых часто используемых методов при анализе данных. В pandas за группировку отвечает метод .groupby.

**Сводные таблицы в pandas**

В pandas сводные таблицы строятся через метод .pivot\_table. За основу возьмём всё тот же пример с Титаником.

**Анализ временных рядов**

В pandas очень удобно анализировать временные ряды.

**Алгоритмы поиска**

## **Линейный поиск**

**Линейный поиск** — это один из самых простых и понятных алгоритмов поиска.

Суть алгоритма заключается в том*,* чтобы перебрать массив и вернуть индекс первого вхождения элемента, когда он найден:

Временная сложность линейного поиска равна ***O(n)***. Это означает, что время, необходимое для выполнения, увеличивается с увеличением количества элементов в нашем входном списке.

Линейный поиск хорошо подходит для тех случаев, когда нам нужно найти первое вхождение элемента в несортированной коллекции. Это связано с тем, что он не требует сортировки коллекции перед поиском (в отличие от большинства других алгоритмов поиска).

## **Бинарный поиск**

Бинарный поиск работает по принципу [«разделяй и властвуй»](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D0%B4%D0%B5%D0%BB%D1%8F%D0%B9_%D0%B8_%D0%B2%D0%BB%D0%B0%D1%81%D1%82%D0%B2%D1%83%D0%B9_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)). Он быстрее, чем линейный поиск, но требует, чтобы массив был отсортирован перед выполнением алгоритма.

На каждой итерации алгоритм выполняет одно из следующих действий:

* Возврат индекса текущего элемента.
* Поиск в левой половине массива.
* Поиск в правой половине массива.

Мы можем выбрать только одно действие на каждой итерации. Также на каждой итерации наш массив делится на две части.

Одним из недостатков бинарного поиска является то, что если в массиве имеется несколько вхождений элемента, он возвращает индекс не первого элемента, а ближайшего к середине.

## **Jump Search**

**Jump Search** похож на бинарный поиск тем, что он также работает с отсортированным массивом и использует аналогичный подход «разделяй и властвуй» для поиска по нему.

Его можно классифицировать как усовершенствованный алгоритм линейного поиска, поскольку он зависит от линейного поиска для выполнения фактического сравнения при поиске значения.

В заданном отсортированном массиве мы ищем не постепенно по элементам массива, а скачкообразно

С каждым прыжком мы сохраняем предыдущее значение и его индекс.

Поскольку это сложный алгоритм, давайте рассмотрим пошаговое вычисление:

* Jump search сначала определит размер прыжка путем вычисления *math.sqrt(len(lys))*. Поскольку у нас 9 элементов, размер прыжка будет √9 = 3.
* Далее мы вычисляем значение переменной *right*. Оно рассчитывается как минимум из двух значений: длины массива минус 1 и значения *left + jump*, которое в нашем случае будет 0 + 3 = 3. Поскольку 3 меньше 8, мы используем 3 в качестве значения переменной *right*.
* Теперь проверим, находится ли наш искомый элемент 5 между *lys[0] и lys[3]*. Поскольку 5 не находится между 1 и 4, мы идем дальше.
* Затем мы снова делаем расчеты и проверяем, находится ли наш искомый элемент между *lys[3] и lys[6]*, где 6 — это *3 + jump*. Поскольку 5 находится между 4 и 7, мы выполняем линейный поиск по элементам между *lys[3] и lys[6]* и возвращаем индекс нашего элемента:

## **Поиск Фибоначчи**

**Поиск Фибоначчи** — это еще один алгоритм «разделяй и властвуй», который имеет сходство как с бинарным поиском, так и с jump search. Он получил свое название потому, что использует [числа Фибоначчи](https://ru.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D0%BB%D0%B0_%D0%A4%D0%B8%D0%B1%D0%BE%D0%BD%D0%B0%D1%87%D1%87%D0%B8) для вычисления размера блока или диапазона поиска на каждом шаге.

Числа Фибоначчи — это последовательность чисел 0, 1, 1, 2, 3, 5, 8, 13, 21 …, где каждый элемент является суммой двух предыдущих чисел.

Алгоритм работает с тремя числами Фибоначчи одновременно. Давайте назовем эти три числа *fibM, fibM\_minus\_1 и fibM\_minus\_2*. Где *fibM\_minus\_1 и fibM\_minus\_2* — это два числа, предшествующих *fibM* в последовательности:

*fibM = fibM\_minus\_1 + fibM\_minus\_2*

Мы инициализируем значения 0, 1, 1 или первые три числа в последовательности Фибоначчи. Это поможет нам избежать IndexError в случае, когда наш массив *lys* содержит очень маленькое количество элементов.

Затем мы выбираем наименьшее число последовательности Фибоначчи, которое больше или равно числу элементов в нашем массиве *lys*, в качестве значения *fibM*. А два числа Фибоначчи непосредственно перед ним — в качестве значений *fibM\_minus\_1* и *fibM\_minus\_2*. Пока в массиве есть элементы и значение *fibM* больше единицы, мы:

* Сравниваем *val* со значением блока в диапазоне до *fibM\_minus\_2* и возвращаем индекс элемента, если он совпадает.
* Если значение больше, чем элемент, который мы в данный момент просматриваем, мы перемещаем значения *fibM*, *fibM\_minus\_1* и *fibM\_minus\_2* на два шага вниз в последовательности Фибоначчи и меняем индекс на индекс элемента.
* Если значение меньше, чем элемент, который мы в данный момент просматриваем, мы перемещаем значения *fibM*, *fibM\_minus\_1* и *fibM\_minus\_2* на один шаг вниз в последовательности Фибоначчи.

## **Экспоненциальный поиск**

**Экспоненциальный поиск** — это еще один алгоритм поиска, который может быть достаточно легко реализован на Python, по сравнению с jump search и поиском Фибоначчи, которые немного сложны. Он также известен под названиями galloping search, doubling search и Struzik search.

Алгоритм работает следующим образом:

* Определяется диапазон, в котором, скорее всего, будет находиться искомый элемент.
* В этом диапазоне используется двоичный поиск для нахождения индекса элемента.

# **Алгоритмы сортировки**

**Сортировка** — это когда мы упорядочиваем элементы в итерируемом объекте по возрастанию, убыванию или другому критерию.

**Алгоритм сортировки** — это алгоритм для упорядочивания элементов в списке. В случае, когда элемент в списке имеет несколько полей, поле, служащее критерием порядка, называется ключом сортировки. На практике в качестве ключа часто выступает число, а в остальных полях хранятся какие-либо данные, никак не влияющие на работу алгоритма.

**Bogosort** — неэффективный алгоритм сортировки, используемый только в образовательных целях и противопоставляемый другим, более реалистичным алгоритмам. Bogosort является частным случаем алгоритма Лас-Вегас.

Существуют две версии этого алгоритма: детерминированная версия, которая перебирает все перестановки до тех пор, пока не будет получен отсортированный массив, и случайная версия, которая случайным образом переставляет свои входные данные.

## **Сортировка пузырьком**

Этот простой алгоритм выполняет итерации по списку, сравнивая элементы попарно и меняя их местами, пока более крупные элементы не «всплывут» в начало списка, а более мелкие не останутся на «дне».

## **Сортировка выборкой**

Этот алгоритм сегментирует список на две части: отсортированную и неотсортированную. Наименьший элемент удаляется из второго списка и добавляется в первый.

### **Алгоритм**

На практике не нужно создавать новый список для отсортированных элементов. В качестве него используется крайняя левая часть списка. Находится наименьший элемент и меняется с первым местами.

Теперь, когда нам известно, что первый элемент списка отсортирован, находим наименьший элемент из оставшихся и меняем местами со вторым. Повторяем это до тех пор, пока не останется последний элемент в списке.

По мере увеличения значения i нужно проверять меньше элементов.

## **Пирамидальная сортировка**

Также известна как сортировка кучей. Этот популярный алгоритм, как и сортировки вставками или выборкой, сегментирует список на две части: отсортированную и неотсортированную. Алгоритм преобразует второй сегмент списка в структуру данных «куча» (heap), чтобы можно было эффективно определить самый большой элемент.

### **Алгоритм**

Сначала преобразуем список в Max Heap — бинарное дерево, где самый большой элемент является вершиной дерева. Затем помещаем этот элемент в конец списка. После перестраиваем Max Heap и снова помещаем новый наибольший элемент уже перед последним элементом в списке.

Этот процесс построения кучи повторяется, пока все вершины дерева не будут удалены.

## **Сортировка вставками**

Как и сортировка выборкой, этот алгоритм сегментирует список на две части: отсортированную и неотсортированную. Алгоритм перебирает второй сегмент и вставляет текущий элемент в правильную позицию первого сегмента.

### **Алгоритм**

Предполагается, что первый элемент списка отсортирован. Переходим к следующему элементу, обозначим его х. Если х больше первого, оставляем его на своём месте. Если он меньше, копируем его на вторую позицию, а х устанавливаем как первый элемент.

Переходя к другим элементам несортированного сегмента, перемещаем более крупные элементы в отсортированном сегменте вверх по списку, пока не встретим элемент меньше x или не дойдём до конца списка. В первом случае x помещается на правильную позицию.

## **Сортировка слиянием**

Этот алгоритм относится к алгоритмам «разделяй и властвуй». Он разбивает список на две части, каждую из них он разбивает ещё на две и т. д. Список разбивается пополам, пока не останутся единичные элементы.

Соседние элементы становятся отсортированными парами. Затем эти пары объединяются и сортируются с другими парами. Этот процесс продолжается до тех пор, пока не отсортируются все элементы.

### **Алгоритм**

Список рекурсивно разделяется пополам, пока в итоге не получатся списки размером в один элемент. Массив из одного элемента считается упорядоченным. Соседние элементы сравниваются и соединяются вместе. Это происходит до тех пор, пока не получится полный отсортированный список.

Сортировка осуществляется путём сравнения наименьших элементов каждого подмассива. Первые элементы каждого подмассива сравниваются первыми. Наименьший элемент перемещается в результирующий массив. Счётчики результирующего массива и подмассива, откуда был взят элемент, увеличиваются на 1.

## **Быстрая сортировка (Quicksort)**

Этот алгоритм также относится к алгоритмам «разделяй и властвуй». Его используют чаще других алгоритмов, описанных в этой статье. При правильной конфигурации он чрезвычайно эффективен и не требует дополнительной памяти, в отличие от сортировки слиянием. Массив разделяется на две части по разные стороны от опорного элемента. В процессе сортировки элементы меньше опорного помещаются перед ним, а равные или большие —позади.

**Файлы и файловые операции**

**Файлы** - это именованные области хранения данных на постоянном носителе (например, жесткий диск). Они позволяют сохранять, считывать и манипулировать данными, которые могут быть использованы для дальнейшей обработки или анализа.

* **Функционал:**
  + **Сохранение данных:** Файлы позволяют сохранять данные между запусками программы.
  + **Чтение данных:** Программа может считывать данные из файлов для использования.
  + **Манипулирование данными:** Можно обновлять, добавлять или удалять данные в файлах.
  + **Обмен данными:** Файлы позволяют обмениваться данными между различными программами и системами.

**Файловые операции (обзор и примеры Python):**

1. **Открытие файла:**
   * Используется функция open().
   * Пример: f = open('text.txt', 'r')
2. **Чтение из файла:**
   * Метод read():
     + f.read(): Читает весь файл целиком.
     + f.read(n): Читает n символов.
   * Чтение построчно:
     + Использование цикла for: Позволяет перебирать фай построчно.
3. **Запись в файл:**
   * Метод write(): Записывает строку в файл. Возвращает количество записанных символов.
   * Пример: f.write("Текст для записи")
4. **Закрытие файла:**
   * Метод close(): Обязательно нужно закрывать файл после работы с ним.
   * Пример: f.close()
5. **Другие файловые операции (с использованием модуля os и shutil):**
   * **Копирование файла:** shutil.copyfile("source", "destination")
   * **Переименование файла:** os.rename("old\_name", "new\_name")
   * **Удаление файла:** os.remove("filename")
   * **Чтение определенной строки:** Используя linecache.getline() или open("filename").readlines()[line\_number]
   * **Перебор файлов в каталоге:** os.listdir("directory")
   * **Перебор файлов по маске:** glob.glob("directory/\*.extension")
   * **Сравнение файлов:** filecmp.cmp("file1", "file2")

**Работа с разными типами файлов:**

Текст также охватывает работу с различными типами файлов и предоставляет примеры использования соответствующих библиотек:

* **MS Word (.docx):** Использование библиотеки python-docx. Рассмотрены чтение, запись, добавление абзацев, заголовков, изображений, таблиц, изменение стилей.
* **MS Excel (.xlsx, .xls):** Использование библиотек Pandas, openpyxl и pyexcel. Рассмотрены чтение, запись, манипуляции с данными, создание DataFrame.
* **CSV (.csv):** Использование Pandas и модуля csv.
* **JSON (.json):** Использование модуля json. Рассмотрены чтение (методы load и loads), запись (методы dump и dumps).
* **Parquet (.parquet):** Использование библиотеки pyarrow. Рассмотрены чтение и запись. Подчеркнуты плюсы (эффективное сжатие, высокая производительность) и минусы (неудобство для малых данных, затраты на запись).

**Функции. Объявление и определение функции. Формальные и фактические параметры. Способы передачи параметров. Вызов функции.**

**Вызвать функцию** — значит передать ей входные данные, необходимые для выполнения и возвращения результата. Когда вы передаете функции входные данные, это называется передача параметра функции.

Ключевое слово def сообщает Python, что вы определяете функцию. После def вы указываете имя функции; оно должно отвечать тем же правилам, что и имена переменных. Согласно конвенции, в имени функции нельзя использовать заглавные буквы, а слова должны быть разделены подчеркиванием вот\_так.

После скобок ставится двоеточие, а новая строка начинается с отступа в четыре пробела. Любой код с отступом в четыре пробела после двоеточия является телом функции. Ключевое слово return используется для определения значения, которое функция возвращает при вызове.

У функции может быть один параметр, несколько параметров или вообще их не быть. Чтобы определить функцию, не требующую параметров, оставьте круглые скобки пустыми.

Функция может принимать параметры двух типов. Те, что встречались вам до этого, называются обязательными параметрами. Когда пользователь вызывает функцию, он должен передать в нее все обязательные параметры, иначе Python сгенерирует исключение.

В Python есть и другой вид параметров — опциональные. Опциональные параметры определяются с помощью следующего синтаксиса: имя\_функции(имя\_параметра = значение\_параметра). Как и обязательные, опциональные параметры нужно отделять запятыми. Ниже приведен пример функции, в коде которой используется опциональный параметр.

Функции Python — это объекты первого класса. Их можно присваивать переменным, хранить в структурах данных, передавать в качестве аргументов другим функциям и даже возвращать в качестве значений из других функций.

Функции, которые могут принимать другие функции в качестве аргументов, также называются функциями высшего порядка. Они необходимы для функционального стиля программирования.

**Область действия имен в коде**

Существует четыре уровня:

1. **Локальная (Local)** — переменные внутри функции.
2. **Объемлющая (Enclosing)** — переменные внешней функции для вложенных функций (если они есть).
3. **Глобальная (Global)** — переменные, объявленные на уровне модуля.
4. **Встроенная (Built-in)** — встроенные имена Python (print, len и др.).

**Правила:**

* Локальные переменные видны только внутри функции.
* Если переменная не найдена локально, Python ищет её в объемлющей, затем в глобальной и, наконец, во встроенной области.
* Для изменения глобальной переменной внутри функции используется ключевое слово global.
* Для изменения переменной из объемлющей функции (но не глобальной) используется nonlocal.

**Перегрузка функций**

**Перегрузка функций**, по-другому универсальные функции — это распространенный шаблон программирования, называемый множественной диспетчеризацией/отправкой, который, наверное, зарезервирован только для статически типизированных, компилируемых языков. Python является динамически типизированным языком и следовательно, перегрузка функций здесь невозможна, тем не менее, есть простой способ реализовать такое поведение в Python.

В Python нет классической перегрузки функций (как в C++ или Java), но её можно эмулировать несколькими способами:

1. **Аргументы по умолчанию** — функция может принимать разное количество параметров.
2. **\*args и \*\*kwargs** — позволяют передавать переменное число аргументов.
3. **Декоратор @singledispatch** — из модуля functools, позволяет определять разные реализации функции в зависимости от типа аргумента.

**Пример использования @singledispatch:**

* Основная функция помечается декоратором.
* Для каждого типа аргумента регистрируется отдельная реализация через @функция.register(тип).
* Если тип не найден, вызывается базовая реализация.

**Стандартные функции**

Python включает множество встроенных функций, которые можно использовать без импорта:

**Основные категории:**

1. **Математические** — abs(), round(), pow(), sum(), min(), max().
2. **Преобразование типов** — int(), float(), str(), list(), dict(), tuple().
3. **Работа с последовательностями** — len(), sorted(), reversed(), enumerate(), zip().
4. **Ввод-вывод** — print(), input(), open().
5. **Проверки** — isinstance(), type(), callable(), hasattr().
6. **Итераторы и генераторы** — range(), iter(), next().

Эти функции помогают выполнять базовые операции без написания дополнительного кода.

**Функции с переменным числом параметров**

Python позволяет создавать функции, принимающие произвольное количество аргументов:

* 1. **\*args (неименованные аргументы)**

Если поставить \* перед именем, это имя будет принимать не один аргумент, а несколько. Аргументы передаются как кортеж и доступны внутри функции под тем же именем, что и имя параметра, только без \*.

* Принимает любое количество позиционных аргументов.
* Внутри функции args становится **кортежем**.
* Пример: функция для суммирования любого количества чисел.

**2. \*\*kwargs (именованные аргументы)**

* Принимает любое количество именованных аргументов (ключ=значение).
* Внутри функции kwargs становится **словарём**.
* Пример: функция, которая выводит переданные параметры в виде "ключ: значение".

\*args и \*\*kwargs — специальный синтаксис, позволяющий передавать в функцию переменное количество аргументов. При этом, совсем не обязательно использовать имена аргументов args и kwargs.

**Рекурсия и рекурсивные функции.**

Реку́рсия — определение, описание, изображение какого-либо объекта или процесса внутри самого этого объекта или процесса, то есть ситуация, когда объект является частью самого себя. Термин «рекурсия» используется в различных специальных областях знаний — от лингвистики до логики, но наиболее широкое применение находит в математике и информатике.

Функции могут вызывать и другие функции, и даже вызывать сами себя! Рассмотрим это на примере функции вычисления факториала. Хорошо известно, что, 0! = 1, 1! = 1. А как вычислить величину *n!* для большого *n*? Если бы мы могли вычислить величину *(n-1)!*, то тогда мы легко вычислим *n!*, поскольку *n! = n(n-1)!*. Но как вычислить *(n-1)!*? Если бы мы вычислили *(n-2)!*, то мы сможем вычисли и *(n-1)! = (n-1) (n-2)!*. А как вычислить *(n-2)!*? Если бы… В конце концов, мы дойдем до величины 0!, которая равна 1.

Подобный прием (вызов функцией самой себя) называется рекурсией, а сама функция называется **рекурсивной.**

**Функции для обработки строк. Определение и инициализация строк. Функции преобразования строки и чисел.**

| **Функция / Метод** | **Описание** |
| --- | --- |
| str.upper() | Возвращает копию строки, где все символы — в верхнем регистре. |
| str.lower() | Возвращает копию строки, где все символы — в нижнем регистре. |
| str.capitalize() | Возводит в верхний регистр первый символ строки, остальные — в нижний. |
| str.title() | Возводит в верхний регистр первый символ каждого слова. |
| str.strip([chars]) | Удаляет указанные символы chars (по умолчанию пробелы) с начала и конца строки. |
| str.lstrip([chars]) | Удаляет символы chars только с начала строки. |
| str.rstrip([chars]) | Удаляет символы chars только с конца строки. |
| str.split(sep=None, maxsplit=-1) | Разбивает строку по разделителю sep и возвращает список подстрок (до maxsplit раз). |
| sep.join(iterable) | Соединяет элементы iterable строкой‑разделителем sep. |
| str.replace(old, new[, count]) | Возвращает копию строки, в которой первые (или все) вхождения old заменены на new. |
| str.find(sub[, start[, end]]) | Возвращает индекс первого вхождения sub или -1, если не найдено. |
| str.index(sub[, start[, end]]) | Как find(), но при отсутствии подстроки генерирует исключение ValueError. |
| str.startswith(prefix[, start[, end]]) | Проверяет, начинается ли строка с prefix. |
| str.endswith(suffix[, start[, end]]) | Проверяет, заканчивается ли строка на suffix. |
| str.isdigit() | Возвращает True, если все символы строки — цифры. |
| str.isalpha() | Возвращает True, если все символы — буквы (только алфавитные). |
| str.isalnum() | Возвращает True, если все символы — буквы или цифры. |
| str.isspace() | Возвращает True, если все символы — пробельные. |
| str.lower(), str.upper(), str.title() | Функции для преобразования регистра символов в строке. |

**Лямбда‑функции.**

«Лямбда-функции — это анонимные однострочные функции, которые в Python создаются с помощью ключевого слова lambda. Основной синтаксис:

lambda арг1, арг2, … : выражение

Где арг1, арг2 — параметры, а после двоеточия — единственное вычисляемое выражение, результат которого возвращается. В отличие от обычных функций, лямбда-функция не содержит тела с несколькими инструкциями и не требует ключевого слова def.

**Пример**:

# Функция, возводящая число в квадрат

sq = lambda x: x \* x

print(sq(5)) # 25

Лямбда-функции широко применяются в сочетании с функциями высшего порядка: map(), filter(), sorted() и т.п., чтобы кратко определить небольшие операции без объявления полноценной функции.»

**Многофаловые приложения**

Модуль — это файл с расширением .py, содержащий связанные между собой инструкции и определения. Для объединения нескольких модулей в единую программу используют пакеты — каталоги с файлом \_\_init\_\_.py. Чтобы получить доступ к функциям, классам и переменным из другого модуля, применяют инструкцию import или from … import ….

**Классы. Определение, атрибуты, методы классов.**

Класс — это своего рода чертеж или шаблон для создания объектов. Он описывает, какими характеристиками (атрибутами) и поведением (методами) будут обладать объекты, созданные на его основе. Внутри класса определяются его атрибуты, которые хранят различные характеристики класса, и методы - функции класса.

Атрибут – характеристики объекта в программировании — например, цена, производитель или объём оперативной памяти. В классе прописывают, что такие атрибуты есть, а в объектах с помощью методов заполняют эти атрибуты данными.

После создания класса можно определить объекты этого класса. **Объект** – экземпляр класса. Если класс — это чертеж, то объект — это конкретная реализация этого чертежа. Например, если у нас есть класс "Автомобиль", то объекты этого класса могут быть "BMW", "Audi" и т. д.

Методы класса фактически представляют функции, которые определенны внутри класса и которые определяют его поведение.

**Магические методы типа \_\_init\_\_, \_\_str\_\_ и т.д.**

Метод \_\_str\_\_ — это так называемый «магический» (специальный) метод в Python, отвечающий за преобразование объекта в «читаемую» строку.

Конструктор — это специальный метод класса, который автоматически вызывается при создании нового объекта. В Python он обозначается как \_\_init\_\_ и используется для инициализации атрибутов экземпляра.

В языке Python принят стандарт, что методы с именами, начинающимися и заканчивающимися двумя символами подчёркивания (double underscore — часто сокращённо «dunder»), многое значат для интерпретатора. Это не обычные методы, а точки вхождения в «внутренний» механизм языка: создание объекта, арифметика, сравнение, конвертация в строку и т. д.

Благодаря такому оформлению риск случайного столкновения имен минимален: вы не переопределите бездумно метод \_\_add\_\_ или \_\_init\_\_, если назовёте свой метод просто add или init.

**Объекты класса. Создание объектов, доступ к атрибутам объектов, вызов методов объекта.**

Объект — это экземпляр класса. После описания класса его можно «вызвать» как функцию

Создание: name = classname(atr1, atr2, …)

Доступ к атрибуту объекта: print(obj.atr)

Вызов метода объекта: obj.method

**Декораторы**

Декораторы — это специальные функции (или вызываемые объекты), которые принимают другую функцию или метод и возвращают модифицированный или обёрнутый вариант без изменения исходного кода оборачиваемого объекта. Их применяют для отложенного выполнения обёртывания, добавления логирования, контроля доступа, измерения времени выполнения и других аспектов.

**Наследование**

**Наследование** — это механизм ООП, позволяющий одному классу (подклассу) наследовать свойства и методы другого класса (родительского). Наследование упрощает повторное использование кода и расширение функциональности без модификации исходного класса.

**Определение**:  
«Наследование (Inheritance) — механизм, позволяющий одному классу (подклассу) унаследовать свойства и методы другого класса (родительского класса). Наследование способствует повторному использованию кода и упрощает его поддержку» .

**Виды наследования**:

1. **Простое (одноуровневое)** — один подкласс наследует один родительский класс.
2. **Множественное** — подкласс наследует сразу несколько родительских классов.
3. **Многоуровневое** — цепочка классов, где A → B → C.
4. **Иерархическое** — один родительский класс и несколько подклассов.
5. **Гибридное** — сочетание различных видов наследования.

**Инкапсуляция**

«Инкапсуляция — один из ключевых принципов объектно‑ориентированного программирования, который помогает управлять доступом к данным и методам объектов. В Python инкапсуляция реализуется через использование модификаторов доступа:

Одиночное подчеркивание в начале имени атрибута говорит о том, что переменная или метод не предназначен для использования вне методов класса, однако атрибут доступен по этому имени.

Двойное подчеркивание в начале имени атрибута даёт большую защиту: атрибут становится недоступным по этому имени. Однако полностью это не защищает, так как атрибут всё равно остаётся доступным под именем \_ИмяКласса\_\_ИмяАтрибута:

Для контроля доступа и валидации обычно применяют **геттеры** и **сеттеры**, а в Python — декораторы @property и @<property>.setter, обеспечивающие безопасное чтение и запись приватных атрибутов.»

**Полиморфизм**

Полиморфизм — это одна из ключевых концепций объектно-ориентированного программирования (ООП). Слово "полиморфизм" происходит от греческих слов "поли" (много) и "морф" (форма), что буквально означает "много форм". В контексте программирования полиморфизм позволяет методам и функциям работать с объектами разных типов, не зная заранее, какого типа объект будет передан.

Полиморфизм в программировании помогает разработчикам создавать более универсальные и адаптируемые решения. Например, представьте, что у вас есть несколько классов, представляющих различные виды животных, и каждый из этих классов имеет метод speak. Благодаря полиморфизму, вы можете вызвать метод speak для любого объекта животного, не зная его конкретного типа, и получить правильный результат. Это позволяет писать более общий и переиспользуемый код, что особенно важно в крупных проектах.

Полиморфизм также можно использовать с операторами. Например, оператор + может быть перегружен для выполнения различных задач в зависимости от типов операндов. Рассмотрим пример с классами Vector2D и Vector3D

В этом примере оператор + перегружен для классов Vector2D и Vector3D, что позволяет складывать векторы различных размерностей. Это еще один пример полиморфизма, где один и тот же оператор выполняет разные действия в зависимости от типов операндов.

## **Преимущества использования полиморфизма**

Полиморфизм имеет множество преимуществ, среди которых:

* Упрощение кода: Полиморфизм позволяет писать более общий и менее детализированный код, который легче поддерживать и расширять. Это особенно полезно в больших проектах, где изменение одного компонента может затронуть множество других частей кода.
* Гибкость: Код становится более гибким и может работать с объектами разных типов без изменения основной логики. Это позволяет легко адаптировать существующий код к новым требованиям и условиям.
* Расширяемость: Легко добавлять новые классы и методы, не изменяя существующий код. Это упрощает процесс добавления новых функций и возможностей в программу.
* Повторное использование кода: Полиморфизм способствует повторному использованию кода, так как один и тот же метод или функция может работать с различными типами объектов. Это уменьшает количество дублирующегося кода и облегчает его поддержку.
* Улучшение читаемости кода: Полиморфизм делает код более читаемым и понятным, так как он позволяет использовать более абстрактные и обобщенные конструкции. Это облегчает понимание и поддержку кода, особенно в больших проектах.

## **Формы полиморфизма**

**Полиморфизм подтипов.** Это полиморфность «по умолчанию»: когда в ООП говорят о полиморфизме, обычно имеют в виду его. Выше мы рассказывали именно про такой тип. Это возможность использовать одни и те же команды, или интерфейсы, для разных сущностей — подтипов.

**Параметрический полиморфизм.** Его еще называют обобщенным полиморфизмом. В нем для команды не имеет значения, какую сущность ей прислали: для всех возможных классов будет использоваться один код. Такой полиморфизм считается «истинным» и делает код универсальнее, но реализовать его сложнее.

**Полиморфизм ad hoc.** Этот вид полиморфизма еще называют специализированным: идея ad hoc — разный код при одинаковом названии. Часто такой полиморфизм реализуют с помощью перегрузки методов: несколько раз пишут метод с одним и тем же названием, но разным кодом.

**Абстракция**

Абстракция (от лат. abstractio — отвлечение) — это выделение главных, наиболее значимых характеристик объекта и отвлечение от второстепенных.  
В объектно-ориентированном программировании абстракция позволяет скрыть детали реализации и представить пользователю только необходимый интерфейс.

В Python абстракция реализуется с помощью модуля abc (Abstract Base Classes). Этот модуль предоставляет возможность создавать абстрактные классы, которые нельзя инстанцировать напрямую, и абстрактные методы, которые обязаны быть переопределены в подклассах.

Для определения абстрактного класса необходимо:

* Наследовать его от abc.ABC;
* Использовать декоратор @abstractmethod для методов, которые должны быть реализованы в подклассах.

Абстракция позволяет думать на более высоком уровне и упрощает взаимодействие с системой, фокусируясь на том, что объект делает, а не как он это делает

**Исключения. Механизм обработки исключений. Классы для обработки исключений**.

Исключения — это механизм обработки ошибок, возникающих во время выполнения программы. В Python он позволяет перехватывать и обрабатывать ошибки, предотвращая аварийное завершение программы.

**Механизм обработки исключений** реализуется с помощью конструкции try–except–else–finally

**Блоки конструкции**:

* try: содержит потенциально «опасный» код;
* except: ловит исключения, можно обрабатывать конкретные типы (например, ValueError, TypeError);
* else: выполняется, если в блоке try не возникло исключений;
* finally: выполняется всегда — для освобождения ресурсов, закрытия файлов и т.п.

**Иерархия классов исключений**  
Все встроенные исключения наследуются от базового класса BaseException. Основной класс — Exception. От него наследуются, например:

* ArithmeticError
  + ZeroDivisionError
  + OverflowError
  + FloatingPointError
* ValueError
* TypeError
* IOError / OSError
* FileNotFoundError

Иногда встроенных типов ошибок недостаточно. Тогда можно создать **пользовательское исключение**, унаследовав его от класса Exception

*class MyCustomError(Exception):*

*pass*

*raise MyCustomError(\"Что-то пошло не так\"*

Это позволяет более точно контролировать поведение программы при ошибках, особенно в больших проектах.

**Параллельное программирование. GIL.**

GIL — глобальная блокировка интерпретатора, которая обеспечивает одновременное выполнение только одного потока байт-кода Python в интерпретаторе CPython.  
Это значит, что даже если программа создала несколько потоков, интерпретатор может исполнять только один поток в любой момент времени.

Такое ограничение связано с тем, что в CPython объекты не являются потокобезопасными, и GIL обеспечивает безопасность работы со встроенными типами данных без необходимости синхронизации на уровне каждого объекта.

Однако это ведёт к тому, что многопоточность в Python эффективно работает только для операций ввода/вывода, где потоки часто простаивают в ожидании (например, сетевые запросы, работа с файлами).

Для вычислительно тяжёлых задач (CPU-bound) рекомендуется использовать многопроцессорный подход — каждый процесс имеет собственный GIL, и таким образом можно использовать все доступные ядра процессора.

Параллельное программирование — это подход к разработке программного обеспечения, при котором задачи выполняются одновременно. Оно позволяет использовать преимущества современных многоядерных процессоров и ускоряет выполнение ресурсоёмких операций.

В Python есть несколько способов реализации параллелизма:

1. **Многопоточность (threading)** — позволяет запускать несколько потоков в рамках одного процесса. Однако из-за GIL в CPython многопоточность не даёт настоящего параллелизма в задачах, связанных с вычислениями, но может быть эффективной в задачах ввода/вывода.
2. **Многопроцессорность (multiprocessing)** — каждый процесс имеет свою собственную память и интерпретатор, включая отдельный GIL. Это позволяет выполнять несколько процессов параллельно на разных ядрах процессора.
3. **Асинхронное программирование (asyncio)** — не использует потоки или процессы. Вместо этого применяется кооперативная многозадачность: функции приостанавливаются в ожидании операции (например, чтение файла), и в это время может выполняться другой код.

Выбор подхода зависит от типа задачи:

* для операций с файлами и сетью — асинхронность или потоки,
* для сложных вычислений — процессы.

**Многопоточное и многопроцессорное программирование**

«При разработке программ, требующих высокой производительности или асинхронного выполнения задач, важно понимать разницу между многопоточностью и многопроцессорностью.

🔹 **Многопоточность (threading)**

Многопоточность позволяет программе выполнять несколько операций (потоков) параллельно в рамках одного процесса. Потоки разделяют одну и ту же память и глобальные переменные.

В Python многопоточность реализуется с помощью модуля threading.

Однако из-за наличия GIL (Global Interpreter Lock) только один поток может исполняться в каждый конкретный момент времени. Это означает, что в задачах, требующих интенсивных вычислений, многопоточность в Python **не даёт прироста производительности**.

Но в задачах ввода-вывода (I/O-bound) многопоточность даёт хорошие результаты:

* загрузка файлов с интернета,
* общение с базами данных,
* сетевое взаимодействие,
* ожидание ответа от пользователя.

🔹 **Многопроцессорность (multiprocessing)**

В отличие от многопоточности, многопроцессорность позволяет запускать несколько **отдельных процессов**, каждый из которых имеет:

* свою собственную память,
* свой собственный интерпретатор Python,
* **собственный GIL**,

...и может выполняться независимо и **реально параллельно** на многоядерных системах.

Python предоставляет модуль multiprocessing, который по синтаксису напоминает threading, но запускает отдельные процессы.

Многопроцессорность позволяет реализовать настоящую параллельную обработку данных и особенно полезна в **CPU-bound задачах** — например, при сложных математических вычислениях, обработке изображений, машинном обучении и других ресурсоёмких операциях.

➕ **Преимущества многопроцессорности**:

* использование всех доступных ядер процессора;
* независимое выполнение кода без блокировок;
* избежание конфликтов при работе с памятью.

➖ **Недостатки**:

* процессы требуют больше памяти, чем потоки;
* выше накладные расходы на создание и коммуникацию между процессами.

➤ Для обмена данными между процессами используют очереди (Queue), каналы (Pipe) и общий менеджер памяти (Manager).

➤ Выбор между threading и multiprocessing зависит от конкретной задачи:

* если задача зависит от времени ожидания (I/O-bound) — **подходят потоки**;
* если задача требует большого количества вычислений (CPU-bound) — **лучше процессы**.»

**Асинхронное программирование**

«Асинхронное программирование — это подход, при котором функции могут **приостанавливать своё выполнение**, чтобы дать другим функциям возможность выполниться. Это особенно эффективно в задачах, связанных с **вводом-выводом**, где основное время занимает ожидание (например, ответа от сервера или чтения файла).

В Python асинхронность реализуется с помощью ключевых слов async и await и модуля asyncio.

🔹 Основные понятия:

* **Корутина** — функция, объявленная с async def, которую можно приостанавливать и возобновлять.
* **await** — используется для ожидания выполнения корутины или асинхронной операции.
* **событийный цикл** — механизм, который управляет выполнением корутин.

*import asyncio*

*async def say\_hello():*

*print("Привет")*

*await asyncio.sleep(1)*

*print("Мир")*

*asyncio.run(say\_hello())*

Здесь asyncio.sleep(1) — асинхронная задержка, которая не блокирует другие задачи.

Асинхронное программирование не требует создания потоков или процессов и хорошо подходит для задач, где основная нагрузка связана с ожиданием, а не с вычислениями.

Преимущества:

высокая эффективность в I/O-bound задачах (работа с сетью, базами данных, файлами);

меньшая нагрузка на память по сравнению с потоками и процессами;

простота масштабирования.

Недостатки:

требует другой архитектуры кода (всё должно быть async);

не подходит для вычислительно тяжёлых задач (где лучше использовать multiprocessing).

Асинхронность позволяет добиться высокой производительности без создания потоков и процессов и может быть эффективно использована в веб‑разработке, микросервисах, телеграм‑ботах, API‑интерфейсах и серверной логике.

**Тестирование. Назначение. Виды.**

**Тестирование** — это проверка соответствия программы требованиям, осуществляемая путём наблюдения за её работой в специальных, искусственно созданных ситуациях, выбранных определённым образом.

## **Виды тестирования**

* Ручное – когда тестировщик выступает в роли конечного пользователя и проверяет программу своими действиями
* Автоматическое – когда для тестирования программы тестировщик использует библиотеки для тестирования и пишет специализированные тестовые программы.

## **Уровни автоматического тестирования**

Тестирование для разработчика состоит из написания тестов. Тест – это такой же программный код, который пишется аналогично коду для реализации бизнес-логики. Тесты проверяют сценарии работы программы (test-case).

Тест-кейсы встречаются самые различные, один от другого может резко отличаться. По желанию можно тестировать ВСЕ возможные и невозможные ситуации. Однако стоит соблюдать адекватность и покрывать код тестами ровно настолько, насколько требуется для уверенного понимания, что бизнес-логика работает как задумано.

Негласное правило – если на участке кода проявилась ошибка, то стоит написать тест на этот случай.

Тестирование смело делится на несколько уровней глубины. Наиболее показательная классификация по уровням тестирования в данном случае такая:

* Системное тестирование – тестирование полностью интегрированного программного приложения
* Интеграционное тестирование – тестирование интегрированных групп программных модулей
* Модульное тестирование или юнит-тестирование – тестирование отдельных модулей исходного кода приложения

Когда говорят "тестирование", не выделяя конкретный тип, то говорят скорее всего о модульном тестировании.

**Ручное тестирование**

Ручное тестирование — это способ тестирования, при котором человек вручную проверяет работу приложения без использования специальных автоматизированных инструментов.

🔹 Оно применяется:

* на ранних этапах разработки,
* для проверки пользовательского интерфейса,
* при работе с нестандартными сценариями,
* когда автоматизация нерентабельна или невозможна.

🔹 Преимущества:

* Гибкость в тестировании;
* Возможность быстро отреагировать на непредвиденные ошибки;
* Хорошо подходит для проверок визуального интерфейса и пользовательского опыта (UX/UI).

🔹 Недостатки:

* Низкая воспроизводимость результатов;
* Зависимость от человеческого фактора (возможны пропуски);
* Затраты времени на повторные тесты.

Ручное тестирование часто используется в комбинации с автоматизированным тестированием: сначала продукт проверяется вручную, а затем стабилизированные части переводятся на автотесты.

**Автоматизация тестирования**

Автоматизация тестирования — это процесс использования специализированных программных средств и скриптов для автоматической проверки правильности работы приложения.

Она применяется там, где необходимо регулярно выполнять большое количество однотипных тестов, особенно в условиях частых обновлений.

**Преимущества автоматизации**:

* Существенное ускорение процесса тестирования;
* Высокая точность и повторяемость;
* Снижение влияния человеческого фактора;
* Возможность интеграции в конвейер CI/CD (непрерывной интеграции и доставки).

**Недостатки**:

* Требуются начальные затраты на разработку и поддержку автотестов;
* Не всегда подходит для UI/UX и визуальных проверок.

**Инструменты автоматизации**:

* Для модульного тестирования: unittest, pytest, nose;
* Для UI-тестирования: Selenium, Playwright, Cypress;
* Для API: Postman, REST Assured, pytest + requests;
* Для интеграции в CI: Jenkins, GitLab CI, GitHub Actions.

Обычно автоматизации подвергаются стабильные участки системы, которые часто регрессируют или требуют постоянной проверки.

Важно: автоматизация не заменяет ручное тестирование полностью, а дополняет его — особенно при регрессионных и нагрузочных проверках.