**1.** В Python функции — это блоки кода, которые выполняют определенную задачу. Они позволяют организовать код, сделать его более читаемым и многократно использовать.

**2.** Вложенные функции — это функции, определенные внутри другой функции. Они доступны только изнутри внешней функции и не видны за её пределами.

return используется для возврата значения из функции. Если return не используется, функция возвращает None.

yield используется для создания генератора — объекта, который генерирует значения по запросу. При каждом вызове next() генератор возвращает значение, которое было после последнего вызова yield.

**Ключевые отличия yield от return:**

**return завершает выполнение функции, `yield` приостанавливает его.**

return возвращает значение, yield возвращает генератор.

**3.** Функция может возвращать одно простое значение, используя инструкцию return.

В Python можно возвращать несколько значений из функции, используя кортежи.

**Можно возвращать списки, словари и другие типы данных, помимо кортежей.** Функция может не возвращать ничего, в этом случае она вернет None.

**4.** В Python, аргументы — это значения, которые передаются в функцию во время её вызова, а параметры — это переменные, которые принимают эти значения внутри функции.  
\*args позволяет функции принимать произвольное количество позиционных аргументов. Эти аргументы будут собраны в кортеж.

\*\*kwargs позволяет функции принимать произвольное количество именованных аргументов. Эти аргументы будут собраны в словарь.

**`**args и kwargs могут использоваться вместе в одной функции. \* **args` должен идти перед `kwargs`.** Порядок позиционных аргументов важен, а именованные аргументы могут быть переданы в любом порядке.

**5.** Позиционные аргументы в Python — это аргументы, которые передаются в функцию в определенном порядке. Их позиция определяет, какой параметр функции получит какое значение.

Порядок аргументов: При вызове функции порядок позиционных аргументов должен совпадать с порядком параметров в определении функции.

Количество аргументов: Количество позиционных аргументов, переданных при вызове функции, должно соответствовать количеству параметров в определении функции.

**6.** Рекурсия - это техника, когда функция вызывает саму себя внутри своего определения. Это позволяет решать задачи путем разбивки их на подзадачи того же типа.

Преимущества рекурсии:

* Чёткость: Рекурсивные решения могут быть более понятными и лаконичными для некоторых задач.
* Структурирование данных: Рекурсия естественна для задач, связанных с деревьями, графами и другими иерархическими структурами.

Недостатки рекурсии:

* Сложность отладки: Отслеживание вызовов функции может быть сложным.
* Ограничения по стеку: Слишком глубокая рекурсия может привести к переполнению стека (Stack Overflow).
* Эффективность: Рекурсивные решения могут быть менее эффективными, чем итеративные, из-за дополнительного использования памяти для хранения стека вызовов.

**7.** Модуль в Python - это файл, содержащий Python-код. Он может содержать функции, классы, переменные и другие определения, которые можно использовать в других файлах. Модули позволяют организовать код, разбить его на логические части и повторно использовать его в разных проектах.  
Чтобы использовать код из модуля, необходимо импортировать его в текущий файл с помощью команды import.

Варианты импорта:

- Импорт всего модуля: import math

- Импорт определенных элементов из модуля: from math import pi, sqrt

- Импорт всего модуля с псевдонимом: import math as m

- Импорт определенных элементов из модуля с псевдонимом: from math import pi as a

Плюсы:

* **Организация кода: Делит большой проект на логические части, что упрощает его понимание и поддержку.**
* Повторное использование: Модули можно повторно использовать в разных проектах. **Разделение ответственности: Разные команды могут работать над разными модулями независимо друг от друга.**
* Снижение дублирования кода: Позволяет избежать дублирования кода, что повышает эффективность разработки.

**8.** Модуль `math` в Python предоставляет набор математических функций и констант.

Примеры:

`math.pi`: Константа, представляющая число пи (π) с высокой точностью.

`math.sqrt(x)`: Возвращает квадратный корень из числа `x`.

`math.sin(x)`: Возвращает синус угла `x`, выраженного в радианах.

`math.cos(x)`: Возвращает косинус угла `x` в радианах.

`math.tan(x)`: Возвращает тангенс угла `x` в радианах.

`math.ceil(x)`: Возвращает наименьшее целое число, не меньшее `x`.

`math.floor(x)`: Возвращает наибольшее целое число, не большее `x`.

`math.pow(x, y)`: Возвращает результат возведения `x` в степень `y`.

`math.log(x, base)`: Возвращает логарифм `x` по основанию `base`.

`math.exp(x)`: Возвращает значение `e` в степени `x`, где `e` — основание натурального логарифма.

`math.degrees(x)`: Преобразует угол из радиан в градусы.

`math.radians(x)`: Преобразует угол из градусов в радианы.

**9.** Модуль `cmath` в Python предоставляет функции для работы с комплексными числами. Комплексное число - это число, которое может быть представлено в виде `a + bi`, где `a` и `b` - действительные числа, а `i` - мнимая единица, определяемая как `i² = -1`.

complex(3, 4) – Создает комплексное число 3 + 4i

действительная\_часть = z.real

мнимая\_часть = z.imag

`cmath.polar(z)`: Возвращает модуль и аргумент комплексного числа `z` в виде кортежа.

`cmath.rect(r, phi)`: Возвращает комплексное число, заданное модулем `r` и аргументом `phi` в радианах.

`cmath.sqrt(z)`: Возвращает квадратный корень из комплексного числа `z`.

`cmath.exp(z)`: Возвращает значение `e` в степени `z`, где `e` — основание натурального логарифма.

`cmath.log(z)`: Возвращает натуральный логарифм комплексного числа `z`.

**10.** Модуль `unittest` в Python предоставляет набор инструментов для написания тестов и проверки корректности кода.

Модуль `unittest` в Python предоставляет набор инструментов для написания тестов и проверки корректности кода.

Структура:

1. Тестовый класс: Тесты обычно организованы в классах, наследуемых от `unittest.TestCase`. Каждый тестовый метод должен начинаться с `test\_`.

2. Методы утверждений: Внутри тестовых методов используются методы утверждений (`assert...`) для проверки ожидаемых результатов.

### Основные методы утверждений:

`assertEqual(a, b)`: Проверяет, равны ли `a` и `b`.

`assertTrue(x)`: Проверяет, что `x` истинно.

`assertIsNone(x)`: Проверяет, что `x` равно `None`.

`assertIn(a, b)`: Проверяет, что `a` содержится в `b`.

`assertRaises(exception, callable, \*args, \*\*kwargs)`: Проверяет, что вызываемая функция `callable` с аргументами `\*args` и `\*\*kwargs` генерирует исключение `exception`.

Написание тестов является важной частью процесса разработки программного обеспечения. Это позволяет гарантировать качество и корректность кода.

**11.** Модуль `fractions` в Python позволяет работать с рациональными числами (дробями) в точном виде, без преобразования их в числа с плавающей точкой, которые могут привести к ошибкам округления.

Fraction(3, 4) # Создает дробь ¾

Fraction("1/2") # Создает дробь ½

Fraction(5) # Создает дробь 5/1

`Fraction.numerator`: Возвращает числитель дроби.

`Fraction.denominator`: Возвращает знаменатель дроби.

`Fraction.limit\_denominator(max\_denominator)`: Находит дробь, максимально близкую к текущей, но с знаменателем не больше `max\_denominator`.

Преимущества:

Точность: Сохраняет точное представление рациональных чисел без ошибок округления.

Чтение и понимание: Дроби легче читать и интерпретировать, чем числа с плавающей точкой.

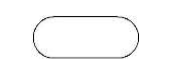
Программирование: Обеспечивает более точные вычисления в математических задачах.

**12.** Создание и подключение собственного модуля (Легко)

**13.** Линейные алгоритмы - это алгоритмы, которые выполняются последовательно, шаг за шагом, без каких-либо циклов или ветвлений.

Блок-схемы - это графическое представление алгоритмов, которое используется для визуализации и понимания их логики. В блок-схемах каждый шаг алгоритма представлен блоком, а связь между шагами показана стрелками.

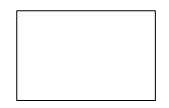
Блок начала-конца:



Блок ввода-вывода данных (отображает список вводимых и выводимых переменных):

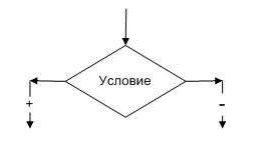


Арифметический блок (отображает арифметическую операцию/группу операций):



**14.** Алгоритмы с ветвлением - это алгоритмы, которые выполняют разные действия в зависимости от определенного условия. Они позволяют программе принимать решения и изменять свой ход выполнения в зависимости от входных данных.

Условный блок (позволяет описать условие). Алгоритмы с таким блоком используются при графической визуализации алгоритмов с ветвлением:



**15.** Конструкция if-elif-else

**16.** Двумерные массивы (или матрицы) - это структуры данных, которые представляют собой таблицу, состоящую из строк и столбцов. Каждый элемент в таблице имеет уникальный индекс, который состоит из двух частей: \*\*индекса строки\*\* и \*\*индекса столбца\*\*.

**17.** Алгоритм — это **набор инструкций, описывающих порядок действий исполнителя для достижения результата решения задачи за конечное число действий.**

**Свойства алгоритма:**

1. **Дискретность.** Алгоритм должен представлять процесс решения задачи как последовательное выполнение некоторых простых шагов.
2. **Детерминированность (определённость).** В каждый момент времени следующий шаг работы однозначно определяется состоянием системы.
3. **Понятность.** Алгоритм должен включать только те команды, которые доступны исполнителю и входят в его систему команд.
4. **Завершаемость (конечность).** При корректно заданных исходных данных алгоритм должен завершать работу и выдавать результат за конечное число шагов.
5. **Массовость (универсальность).** Алгоритм должен быть применим к разным наборам исходных данных.
6. **Результативность.** Алгоритм должен заканчиваться определёнными результатами.

**18.** Сложность алгоритма - это мера количества ресурсов (времени или памяти), которые требуется алгоритму для выполнения задачи в зависимости от размера входных данных.

О-нотация - это математический инструмент, который используется для описания асимптотической сложности алгоритма, то есть его поведения при увеличении размера входных данных.

Основные классы сложности:

- O(1) (Константная сложность): Время выполнения алгоритма не зависит от размера входных данных.

- O(log n) (Логарифмическая сложность): Время выполнения алгоритма растет пропорционально логарифму размера входных данных.

- O(n) (Линейная сложность): Время выполнения алгоритма растет линейно пропорционально размеру входных данных.

- O(n log n) (Логарифмически-линейная сложность): Время выполнения алгоритма растет как произведение размера входных данных и его логарифма.

- O(n^2) (Квадратичная сложность): Время выполнения алгоритма растет пропорционально квадрату размера входных данных.

- O(2^n) (Экспоненциальная сложность): Время выполнения алгоритма растет экспоненциально по отношению к размеру входных данных.

**19.** Односвязный список — это линейная структура данных, в которой элементы (узлы) связаны друг с другом по цепочке. Каждый узел содержит два поля:

**Данные: Хранит информацию, связанную с узлом.**

**Ссылка (указатель):** Ссылается на следующий узел в списке.

Преимущества:

Эффективная вставка/удаление: Вставка и удаление элементов в списке происходит быстро, что делает их подходящими для задач, которые требуют динамического обновления данных.

Недостатки:

Непоследовательный доступ: Доступ к конкретному элементу списка требует прохода по всему списку до нужного узла.

**20.** Двусвязный список - это линейная структура данных, где каждый узел содержит ссылки на два узла: предыдущий и следующий. Это позволяет перемещаться по списку в обоих направлениях, как вперед, так и назад.

Преимущества:

* Двунаправленный доступ: Позволяет эффективно перемещаться по списку в любом направлении.
* Эффективная вставка/удаление: Вставка и удаление узлов могут быть выполнены с помощью небольшого количества операций, что делает двусвязные списки подходящими для задач, где требуется быстрое изменение данных.

Недостатки:

* Дополнительная память: Двусвязные списки требуют больше памяти, чем односвязные, так как каждый узел содержит две ссылки.

**21.** Стек - это структура данных LIFO (Last-In, First-Out), где последний добавленный элемент является первым, который будет извлечен.

Дек - это структура данных, где элементы могут добавляться или удаляться с обоих концов. Он также известен как \*\*двусторонняя очередь\*\*.

Применение:

Стек: Управление историей действий в текстовом редакторе, обработка выражений, рекурсивные вызовы функций.

Дек: Обработка задач, где требуется доступ к элементам с обоих концов, например, буферизация данных.

**22.** Очередь - это структура данных FIFO (First-In, First-Out), где первый добавленный элемент является первым, который будет извлечен.

Очередь с приоритетом - это абстрактный тип данных, в котором элементы упорядочиваются по приоритету. Элемент с наивысшим приоритетом всегда извлекается первым.

Применение:

* Очередь: Обработка задач, где порядок выполнения важен, например, обработка запросов, буферизация данных.
* Очередь с приоритетом: Системы планирования задач, управление ресурсами, обработка событий с разным приоритетом.

**23.** Сортировка вставками. Суть сортировки

1. Перебираются элементы в неотсортированной части массива.
2. Каждый элемент вставляется в отсортированную часть массива на то место, где он должен находиться.

**24.** Пузырьковая сортировка. Алгоритм выглядит так:

1. Берём самый первый элемент массива и сравниваем его со вторым. Если первый больше второго — меняем их местами с первым, если нет — ничего не делаем.
2. Затем берём второй элемент массива и сравниваем его со следующим — третьим. Если второй больше третьего — меняем их местами, если нет — ничего не делаем.
3. Проходим так до предпоследнего элемента, сравниваем его с последним и ставим наибольший из них в конец массива. Всё, мы нашли самое большое число в массиве и поставили его на своё место.
4. Возвращаемся в начало алгоритма и делаем всё снова точно так же, начиная с первого и второго элемента. Только теперь даём себе задание не проверять последний элемент — мы знаем, что теперь в конце массива самый большой элемент.
5. Когда закончим очередной проход — уменьшаем значение финальной позиции, до которой проверяем, и снова начинаем сначала.
6. Так делаем до тех пор, пока у нас не останется один элемент.

**25.** Пирамидальная сортировка (Heap Sort) - это алгоритм сортировки, который использует структуру данных "куча" (heap) для упорядочивания элементов массива.

Как работает алгоритм:

1. Построение кучи: Массив преобразуется в кучу (в данном случае - max-heap, где корневой элемент всегда имеет максимальное значение).
2. Извлечение элементов: Повторяется извлечение корневого элемента (максимального элемента) из кучи и его помещение в конец массива. После каждого извлечения структура кучи восстанавливается.
3. Сортировка: После извлечения всех элементов, массив будет отсортирован в порядке убывания.

**26.** Быстрая сортировка - это эффективный алгоритм сортировки, который использует принцип "разделяй и властвуй". Он работает путем рекурсивного разделения исходного массива на подмассивы, сортировки подмассивов и последующего объединения их в один отсортированный массив.

Как работает алгоритм:

1. Выбор опорного элемента: Из массива выбирается опорный элемент (pivot).
2. Разделение массива: Массив разделяется на две части: элементы, меньшие опорного, и элементы, большие опорного.
3. Рекурсивный вызов: Процедура повторяется рекурсивно для каждой из получившихся частей до тех пор, пока все элементы не будут отсортированы.
4. Объединение частей: Отсортированные подмассивы объединяются в один отсортированный массив.

**27.** Сортировка слиянием (Merge Sort) - это алгоритм сортировки, основанный на принципе "разделяй и властвуй". Он работает путем рекурсивного разделения исходного массива на подмассивы до тех пор, пока каждый подмассив не будет состоять из одного элемента (который по определению уже отсортирован). Затем эти отсортированные подмассивы объединяются (сливаются) в порядке возрастания, создавая полностью отсортированный массив.

Как работает алгоритм:

1. Разделение: Исходный массив рекурсивно делится на две примерно равные половины до тех пор, пока не останется только один элемент в каждой половине.
2. Слияние: Отсортированные подмассивы попарно сливаются в порядке возрастания. При слиянии двух отсортированных подмассивов сравниваются элементы каждого подмассива, и меньший элемент помещается в результирующий отсортированный подмассив. Этот процесс повторяется до тех пор, пока не будут слиты все подмассивы в один отсортированный массив.
3. Объединение: Отсортированные подмассивы объединяются, формируя окончательный отсортированный массив.

**28.** Сортировка подсчетом - это алгоритм сортировки, который подходит для сортировки целых чисел в определенном диапазоне. Он работает путем подсчета количества вхождений каждого элемента в массиве и использования этой информации для создания отсортированного массива.

Как работает алгоритм:

1. Определение диапазона: Определяется минимальное и максимальное значение в массиве.
2. Создание массива подсчета: Создается массив подсчета (count\_array) размером, равным диапазону значений в массиве + 1.
3. Подсчет элементов: Для каждого элемента в исходном массиве увеличивается значение в count\_array на соответствующей позиции.
4. Преобразование в кумулятивные суммы: Каждый элемент в count\_array суммируется с предыдущим, чтобы получить кумулятивную сумму.
5. Создание отсортированного массива: Проходим по исходному массиву и используем count\_array для определения позиции каждого элемента в отсортированном массиве.

Преимущества:

1. Простая реализация: Относительно легко понять и реализовать.
2. Эффективное время работы: О(n + k), где n - количество элементов в массиве, а k - диапазон значений.
3. Стабильная сортировка: Порядок одинаковых элементов сохраняется.

Недостатки:

1. Ограниченный диапазон: Подходит только для сортировки целых чисел в определенном диапазоне.
2. Дополнительная память: Требуется дополнительная память для массива подсчета.
3. Не подходит для больших диапазонов: Если диапазон значений слишком велик, может потребоваться много памяти для массива подсчета.

**29.** Блочная сортировка - это алгоритм сортировки, который работает путем распределения элементов массива по "ведрам" (buckets), а затем сортировки каждого ведра отдельно. После сортировки всех ведер, элементы объединяются в один отсортированный массив.

Как работает алгоритм:

1. Создание ведер: Создается набор ведер, количество которых зависит от размера входного массива и желаемого размера ведра.
2. Распределение элементов: Каждый элемент входного массива помещается в соответствующее ведро. Критерием распределения может быть значение элемента, его остаток от деления на определенное число или любое другое правило.
3. Сортировка ведер: Каждое ведро сортируется с помощью любого другого алгоритма сортировки, например, сортировки вставками или сортировки слиянием.
4. Объединение ведер: Элементы из всех ведер объединяются в один отсортированный массив.

**30.** –

**31.** Устойчивость сортировки (Stability) - это свойство алгоритма сортировки, которое сохраняет относительный порядок элементов с одинаковыми значениями. Другими словами, если в исходном массиве два элемента с одинаковым значением располагаются в определенном порядке, то после сортировки они будут располагаться в том же порядке.

Устойчивые алгоритмы сортировки:

1. пузырьковая
2. вставкой
3. выбором
4. слиянием

Неустойчивые алгоритмы сортировки:

1. быстрая сортировка Хоара
2. пирамидальная

**32.** Линейный поиск (Linear Search) - это простой алгоритм поиска элемента в массиве, который последовательно проверяет каждый элемент массива, пока не найдет искомый элемент или не достигнет конца массива.

Как работает алгоритм:

1. Алгоритм начинает с первого элемента массива.
2. Он сравнивает текущий элемент с искомым элементом.
3. Если элементы совпадают, алгоритм возвращает индекс текущего элемента, то есть позицию, где найден искомый элемент.
4. Если элементы не совпадают, алгоритм переходит к следующему элементу и повторяет шаги 2-3.
5. Если алгоритм доходит до конца массива, а искомый элемент не найден, он возвращает значение `-1`, сигнализируя о том, что элемент отсутствует в массиве.

**33.** Бинарный поиск (Binary Search) - это эффективный алгоритм поиска элемента в отсортированном массиве. Он работает по принципу "разделяй и властвуй", последовательно сужая диапазон поиска, пока не найдет искомый элемент или не установит, что он отсутствует в массиве.

Как работает алгоритм:

1. Алгоритм начинает с проверки среднего элемента массива.
2. Если средний элемент совпадает с искомым элементом, алгоритм возвращает его индекс.
3. Если средний элемент больше искомого элемента, алгоритм продолжает поиск в левой половине массива.
4. Если средний элемент меньше искомого элемента, алгоритм продолжает поиск в правой половине массива.
5. Алгоритм повторяет шаги 2-4, пока не найдет искомый элемент или не достигнет размера массива, равного 1 (то есть не останется элементов для проверки).

**34.** Интерполяционный поиск (Interpolation Search) - это алгоритм поиска элемента в отсортированном массиве, который использует интерполяцию для более точного определения позиции элемента в массиве, чем бинарный поиск.

Как работает алгоритм:

1. Алгоритм определяет позицию элемента в массиве, используя интерполяцию. Интерполяция предполагает, что элементы в массиве распределены равномерно.
2. Алгоритм вычисляет индекс pos по формуле: pos = low + ((x - arr[low]) \* (high - low)) / (arr[high] - arr[low])
3. Алгоритм сравнивает элемент arr[pos] с искомым элементом x.
4. Если элементы совпадают, алгоритм возвращает индекс pos.
5. Если arr[pos] меньше x, алгоритм продолжает поиск в правой половине массива, задавая low = pos + 1.
6. Если arr[pos] больше x, алгоритм продолжает поиск в левой половине массива, задавая high = pos - 1.
7. Алгоритм повторяет шаги 2-6, пока не найдет искомый элемент или не достигнет размера массива, равного 1 (то есть не останется элементов для проверки).