

2022

Теория к экзамену по программированию

Оглавление

1.	Электронно-вычислительная машина. Устройство ЭВМ. Программа. Исходный текст, исполняемый файл.	4
	Структура ЭВМ	5
2.	Схемы алгоритмов.	7
3.	Языки программирования. Классификация.....	10
4.	Язык Python. Структура программы. Лексемы языка.....	11
5.	Типы данных языка Python. Классификация. Скалярные типы данных. Приведение типов. 12	
6.	Операции над скалярными типами данных. Приоритеты операций.....	16
7.	Функции ввода и вывода.	16
8.	Функция вывода. Форматирование вывода.	17
9.	Оператор присваивания. Множественное присваивание.	19
10.	Условный оператор. Полные условные операторы. Неполные условные операторы. Тернарный оператор условия. Примеры использования.....	20
11.	Условные операторы. Множественный выбор. Вложенные операторы условия. Примеры использования.....	21
12.	Операторы цикла. Цикл с условием. Операторы break и continue. Примеры использования.....	22
13.	Операторы цикла. Цикл с итератором. Функция range(). Примеры.....	23
	использования.....	23
14.	Изменяемые и неизменяемые типы данных.	25
15.	Списки. Основные функции, методы, операторы для работы со списками. Срезы.	25
16.	Списки. Создание списков. Списковые включения.....	27
17.	Списки. Основные методы для работы с элементами списка. Добавление элемента, вставки, удаление, поиск.	28
18.	Списки. Основные операции со списками. Поиск минимального элемента. Поиск максимального элемента. Нахождение количества элементов. Нахождение суммы и произведения элементов.....	29
19.	Списки. Использование срезов при обработке списков.	30
20.	Кортежи. Основные функции, методы, операторы для работы с кортежами.	34
21.	Словари. Понятие ключей и значений. Создание словарей. Основные функции, методы, операторы для работы со словарями.	35
22.	Множества. Основные функции, методы, операторы для работы с множествами.....	37
23.	Строки. Основные функции, методы, операторы для работы со строками.	39
	Срезы.....	39
24.	Матрицы. Создание матрицы. Ввод и вывод матрицы. Выполнение операций с 40 элементами матрицы.	40

25.	Матрицы. Квадратные матрицы. Обработка верхне- и нижнетреугольных.....	41
	матриц. Работа с диагональными элементами матрицы.....	41
26.	Отладка программы. Способы отладки.....	43
27.	Подпрограммы. Функции. Создание функции. Аргументы функции.....	43
	Возвращаемое значение.....	43
28.	Функции. Области видимости.	44
29.	Функции. Завершение работы функции. Рекурсивные функции. Прямая и косвенная рекурсия.	45
30.	Функции высшего порядка. Замыкания.....	46
31.	lambda-функции.	48
32.	Аннотации.....	48
33.	Функции map, filter, reduce, zip.....	49
34.	Декораторы.	50
35.	Знак “_”.	51
36.	Модули. Способы подключения.	52
37.	Модуль math. Основные функции модуля. Примеры использования функций.....	53
38.	Модуль time.	55
39.	Модуль random. Работа со случайными числами.	56
40.	Модуль <code>copy</code> . Способы копирования объектов различных типов. “Глубокая” и “мелкая” копии. 58	
41.	Исключения.	59
42.	Файлы. Программная обработка файлов. Понятие дескриптора. Виды файлов.....	60
43.	Файлы. Режимы доступа к файлам.....	62
44.	Файлы. Текстовые файлы. Основные методы для работы.	62
45.	Файлы. Текстовые файлы. Чтение файла. Запись в файл. Поиск в файле.....	63
46.	Файлы. Текстовые файлы. Итерационное чтение содержимого файла.	64
47.	Файлы. Бинарные файлы. Основные методы. Сериализация данных.	64
48.	Файлы. Оператор <code>with</code> . Исключения.	68
49.	Типы данных <code>bytes</code> и <code>bytearray</code> . Байтовые строки. Конвертация различных типов 68	
	в байтовые строки и обратно.	68
50.	Модуль <code>struct</code>	69
51.	Модуль <code>os</code> . Основные функции.....	71
52.	Генераторы.	73
53.	Модуль <code>pintpy</code> . Обработка массивов с использованием данного модуля.	74
54.	Модуль <code>pintpy</code> . Работа с числами и вычислениями.....	77

55.	Модуль matplotlib. Построение графиков в декартовой системе координат. Управление областью рисования.....	79
56.	Модуль matplotlib. Построение гистограмм и круговых диаграмм.....	82
57.	Списки. Сортировка. Сортировка вставками. Сортировка выбором.....	84
58.	Списки. Сортировка вставками. Метод простых вставок. Метод вставок с бинарным поиском. Вставки с барьером. Метод Шелла.....	85
59.	Списки. Сортировка. Обменные методы сортировки. Сортировка пузырьком. Сортировка пузырьком с флагом. Метод шейкер-сортировки.....	87
60.	Списки. Сортировка. Метод быстрой сортировки.....	88

Теория к экзамену по программированию

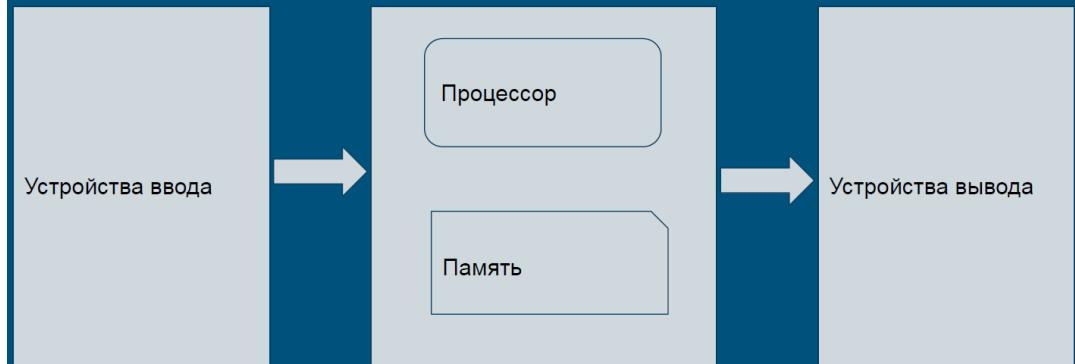
1. Электронно-вычислительная машина. Устройство ЭВМ. Программа. Исходный текст, исполняемый файл.

Компьютер. ЭВМ

Компьютер - устройство, способное выполнять заданную, чётко определённую, изменяемую последовательность операций (численные расчёты, преобразование данных и т. д.)

Электронно-вычислительная машина - основной вид реализации компьютеров, который технически выполнен на электронных элементах

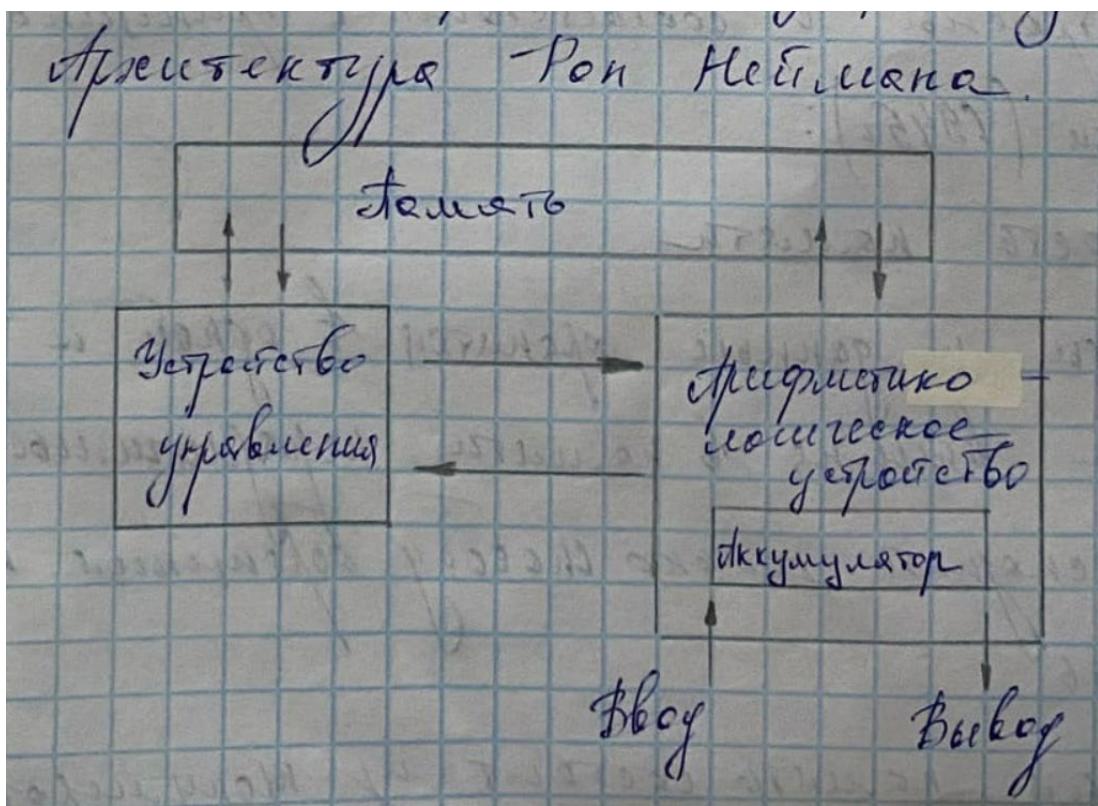
Схема ЭВМ



ЭВМ построены в соответствии с принципами Фон Неймана (1945г):

- Однородность памяти
Команды и данные хранятся в одной и той же памяти и внешне в памяти неразличимы. Их можно распознать только по способу обращения к данным.
- Адресность
Основная память состоит из пронумерованных ячеек, причем процессору в произвольный момент доступна любая ячейка
- Программное управление
Все действия, предусмотренные алгоритмом решения задачи, должны быть представлены в виде программы, состоящей из последовательности команд

Структура ЭВМ



Системная шина – совокупность линий передачи всех видов сигналов, предназначенных для передачи информации между процессором и остальными устройствами ЭВМ

- 1) Шина управления – передача команд
- 2) Шина данных – передача данных
- 3) Адресная шина – передача адресов ячеек памяти

Формы		
Внутренней	Внешней	Магнитные диски
1 Оперативная память ОЗУ	2 Постоянная память ПЗУ	3 Энергозависимая (СМОЗ)
терговависимое	кошечко	жесткие
	диски	диски

- Оперативная память – энергозависимая часть компьютерной памяти, в которой во время работы компьютера хранится выполняемый машинный код (программы), а также входные, выходные и промежуточные данные, обрабатываемые процессором. После выключения ОП очищается.

ОЗУ – RAM (Random Access Memory):

- ✓ SRAM – статическая память
- ✓ DRAM – динамическая память

- Постоянная память – энергозависимая память, используемая для хранения массива неизменяемых данных. Нужна для включения компьютера

- Комплементарный метал-оксидный полупроводник (CMOS, КМОП) – небольшой объем памяти на системной плате, в которой хранятся настройки базовой системы ввода, вывода (BIOS)

Различия оперативной памяти и жесткого диска:

- + ОП хранит информацию пока компьютер включен, иначе стирает.
- + Информация на жестком диске сохраняется вне зависимости от наличия питания.
- + У жесткого диска объем больше (терабайты), чем у ОП (мегабайты).
- + Компьютер загружает на ОП программу с жесткого диска.

Элементарные термины

Процессор – интегральная схема, исполняющая машинные инструкции (код программ), главная часть аппаратного обеспечения компьютера

Машинный код – система команд (набор кодов операций) конкретной вычислительной машины, которая интерпретируется непосредственно процессором. Кодируется в двоичном виде

Файл – поименованное место на диске

Алгоритм – конечная совокупность точно заданных правил решения некоторого класса задач или набор инструкций, описывающих порядок действий исполнителя для решения определённой задачи.

Программа – это логически упорядоченная, строго регламентированная последовательность команд (инструкций) для управления компьютером.

Программа

Исполняемая программа – комбинация компьютерных инструкций и данных, позволяющая аппаратному обеспечению вычислительной системы выполнять вычисления или функции управления

Исходный текст программы – синтаксическая единица, которая соответствует правилам определённого языка программирования, состоящая из определений и операторов или инструкций, необходимых для определённой функции, задачи или решения проблемы

Исполняемый файл

Исполняемый файл - файл, содержащий программу в виде, в котором она может быть исполнена компьютером (то есть в машинном коде).

Получение исполняемых файлов требует выполнения компиляции.

Компилятор - программа для преобразования исходного текста другой программы на определённом языке в объектный модуль (файл с машинным кодом).

2. Схемы алгоритмов.

Схемы алгоритмов

ГОСТ 19.701-90 Схемы алгоритмов,
программ, данных и систем.
Условные обозначения и правила
выполнения

ISO 5807:1985

Блок-схема — распространённый тип схем (графических моделей), описывающих алгоритмы или процессы, в которых отдельные шаги изображаются в виде блоков различной формы, соединённых между собой линиями, указывающими направление последовательности.

Схемы алгоритмов. Символы данных



Данные, носитель которых не определён
(не конкретизирован)



Ввод вручную



Отображающее устройство

Схемы алгоритмов. Символы процесса

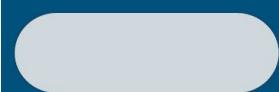


Процесс (операция или группа
операций по обработке данных
любого вида)



Решение: один вход, несколько
альтернативных выходов

Схемы алгоритмов. Специальные символы



Терминатор (начало или конец программы)

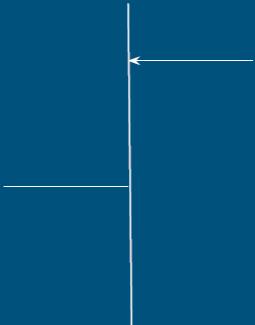
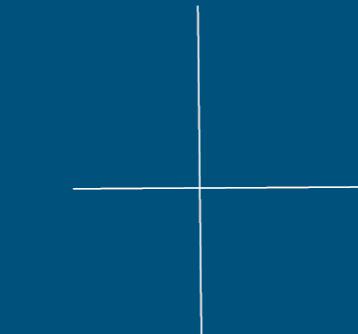


Соединитель - переход в другую часть схемы.
Используется для обрыва линии и продолжения её в другом месте

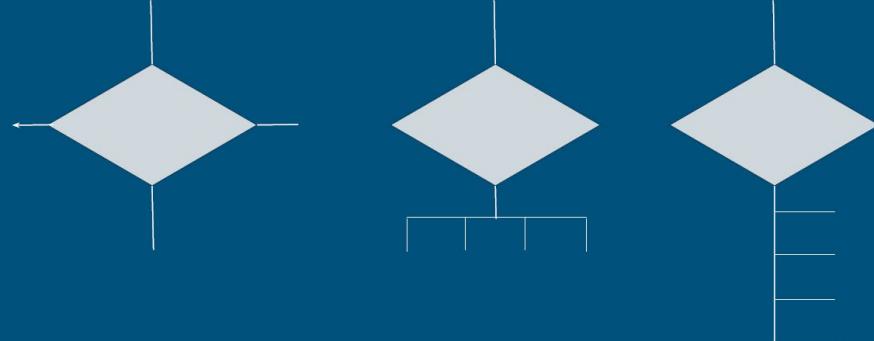
Схемы алгоритмов. Граница цикла



Схемы алгоритмов. Линии



Схемы алгоритмов. Специальные условные обозначения



Схемы алгоритмов. Применение символов

- схема данных
- схема программы
- схема работы системы
- схема взаимодействия программ
- схема ресурсов системы

Другие стандарты схем

- IDEF (I-CAM DEFinition или Integrated DEFinition) - для моделирования широкого спектра сложных систем в различных разрезах
- UML (Unified Modeling Language) - открытый стандарт, использующий графические обозначения для создания абстрактной модели системы
- ARIS (Architecture of Integrated Information Systems) - методология для моделирования бизнес-процессов организаций

3. Языки программирования. Классификация.

Языки программирования

Язык программирования – формальный язык, предназначенный для записи компьютерных программ. Язык программирования определяет набор лексических, синтаксических и семантических правил, определяющих действия, которые выполнит ЭВМ под её управлением.

Классификация языков программирования

- По уровню абстракции от аппаратной части:
 - низкоуровневые
 - высокоуровневые
- По способу выполнения исполняемой программы:
 - компилируемые
 - интерпретируемые
- По парадигме программирования:
 - императивные / процедурные языки
 - аппликативные / функциональные языки
 - языки системы правил / декларативные языки
 - объектно-ориентированные языки

4. Язык Python. Структура программы. Лексемы языка.

Язык программирования Python

Python - высокоуровневый язык программирования общего назначения. Интерпретируемый. Является полностью объектно-ориентированным.

Программа

модули

операторы

выражения

объекты

Лексемы языка Python

Символы алфавита любого языка программирования образуют **лексемы**. По умолчанию - кодировка UTF-8.

Лексема (*token*) – это минимальная единица языка, имеющая самостоятельный смысл. Лексемы формируют базовый словарь языка, понятный компилятору.

Всего существует пять видов лексем:

- ключевые слова (keywords)
- идентификаторы (identifiers)
- литералы (literals)
- операции (operators)
- знаки пунктуации (разделители, punctuators)

Строки программы

Физическая строка исходного файла – это строка заканчивающаяся символом признака конца строки.

Программа Python разделена на несколько логических строк. Логическая строка содержит одну или более физических строк, соединяющихся правилами языка.

Ведущие пробельные символы (пробелы и табуляции) в начале строки используются в Python для определения группы инструкций, как единого целого – составной инструкции или блока.

Комментарии в Python начинаются с символа "#".

Объединение физических строк: явное (через "\") и неявное.

Ключевые слова

```
False  await  else    import  pass
None   break   except  in      raise
True   class   finally  is      return
and    continue for     lambda  try
as     def     from    nonlocal while
assert del    global  not     with
async  elif   if      or      yield
```

5. Типы данных языка Python. Классификация. Скалярные типы данных. Приведение типов.

Основные типы данных Python

- числа
- строки
- списки
- словари
- кортежи
- файлы
- множества
- прочие основные типы
- типы программных единиц
- типы, связанные с реализацией

3.1.2. Скалярные типы

3.1.2.1. Числа

В Python существует 2 категории чисел: целые и вещественные (с плавающей точкой).

3.1.2.1.1. Целое число

Целые числа в Python представлены типом `int`.

```
class int(x=0)
class int(x, base=10)
```

Конструктор класса `int`.

Размер целого числа ограничивается только объемом памяти компьютера. Литералы целых чисел по умолчанию записываются в десятичной системе счисления, но при желании можно использовать и другие (Листинг 3.1.2).

Листинг 3.1.2 - Пример литералов целых чисел

```
>>> 15      # десятичное число
15

>>> 0b1111  # двоичное число
15

>>> 0o17    # восьмеричное число
15

>>> 0xF     # шестнадцатиричное число
15
```

3.1.2.1.2. Вещественное число

Python предоставляет три типа значений с плавающей точкой:

- `float` (двойная точность)
- `complex` (комплексные числа вида `3.5 + 5j`);
- `decimal.Decimal` (большая точность, по умолчанию 28 знаков после запятой).

```
class float([x])
```

Конструктор класса `float`.

Наиболее часто используемый тип `float` представляет числа с плавающей точкой двойной точности, диапазон значений которых зависит от компилятора, применявшегося для компиляции интерпретатора Python. Числа типа `float` записываются с десятичной точкой или в экспоненциальной форме записи (Листинг 3.1.3).

Листинг 3.1.3 - Пример литералов вещественных чисел

```
>>> 5.7          # Точка - разделитель целой и дробной части
5.7

>>> 4.          # Если дробной части нет, ее можно опустить
4.0

>>> -2.5        # Отрицательное вещественное число
-2.5

>>> 8e-4         # Экспоненциальная форма записи
0.0008

>>> 0.1 + 0.2    # Проблема потери точности актуальна для вещественных чисел
0.3000000000000004
```

Для чисел с плавающей точкой существует ряд нюансов:

- в машинном представлении такие хранятся как двоичные числа. Это означает, что одни дробные значения могут быть представлены точно (такие как `0.5`), а другие - только приблизительно (такие как `0.1` и `0.2`, например, их сумма будет равна не `0.3`, а `0.3000000000000004`);
- для представления используется фиксированное число битов, поэтому существует ограничение на количество цифр в представлении таких чисел.

В связи с этим числа типа `float` не могут надежно сравниваться на равенство значений, т.к. имеют ограниченную точность. Проблема потери точности - это не проблема, свойственная только языку Python, а особенность компьютерного представления чисел².

3.1.2.2. Логический тип

Логический тип представлен типом `bool`:

`class bool([x])`

Конструктор класса `bool`.

`True`

`False`

и позволяет хранить 2 значения:

- `True` (Истина / Да / 1);
- `False` (Ложь / Нет / 0).

Операции, которые можно производить над объектами логического типа, указаны в Листинге 3.1.5.

Листинг 3.1.5 - Операции над объектами логического типа в Python

```
>>> x = True
>>> y = False

>>> not x
False

>>> x and y
False

>>> x or y
True

# Python использует "ленивую" модель вычисления: если на каком-то
# этапе результат выражения известен, оно не вычисляется до конца
>>> (4 > 5) and (5 > 2)
False
```

Правила выполнения операций соответствуют логическим таблицам истинности.

3.1.2.3. NoneType

`None`

В Python существует специальное значение `None` типа `NoneType`, обозначающее нейтральное или «нулевое» поведение. Присвоение такого значения ничем не отличается от других: `a = None`, обозначая, что идентификатор `a` задан, но ни с чем не связан.

Наиболее часто используется для защитного программирования - «если что-то не `None`, можно продолжать работу программы».

“Сложность” типов. Приведение типов данных

- Неявное приведение типов:
`123+3.14`
`5.704*1J`
- Явное приведение типов:
 - `int([x])`
 - `int(x, base=10)`
 - `float([x])`
 - `complex([real[, imag]])`

Приведение логического типа данных

- 0, 0.0 - False
- остальные числа - True

```
bool(12.75)  
bool((2*5-10)**14)  
int(True)  
float(False)
```

6. Операции над скалярными типами данных. Приоритеты операций.

Операции над числами

- $x+y$
- $x-y$
- x^y
- x/y
- $x//y$
- $x\%y$
- $x^{**}y$
- $-x$
- $+x$
- $x\|y$
- $x\&y$
- $x\^y$
- $x<<y$
- $x>>y$

Приоритеты:

1. Возведение в степень
2. Умножение, деление, взятие остатка
3. Сложение, вычитание
4. Побитовое И
5. Побитовое исключающее ИЛИ
6. Побитовое ИЛИ

Операции сравнения

Логические операции

and
or
not

<
<=
>
>=
==
!=

Приоритеты операций

1. Арифметические, побитовые (см. предыдущую лекцию)
2. Сравнения
3. Логические

7. Функции ввода и вывода.

Синтаксис:

```
input(prompt)
```

Параметры:

- `prompt` - строка подсказки.

Возвращаемое значение:

- строка `str`, даже если вводятся цифровые значения.

Описание:

Функция `input()` позволяет обеспечить ввод пользовательских данных из консоли. Если передан необязательный аргумент подсказки `prompt`, то он записывается в стандартный вывод без завершающей строки. Затем функция читает строку из ввода и преобразует ее в **СТРОКУ**, убирая завершающий символ строки '\n' и возвращает ее в качестве значения.

Другими словами, все что вводится в консоль при использовании **встроенной функции `input()`** преобразуется в тип `str`. Это происходит в том числе и с числами. Следовательно, числовые данные, перед их использованием необходимо распознавать и преобразовывать к нужным типам.

Синтаксис:

```
print(*objects, sep='', end='\n', file=sys.stdout, flush=False)
```

Параметры:

- `*objects` - объекты Python
- `sep=''` - строка, разделятель объектов. Значение по умолчанию `None`
- `end='\n'` - строка, которой заканчивается поток. Значение по умолчанию `None`
- `file=sys.stdout` - объект, реализующий метод `writeline(string)`. Значение по умолчанию `None`
- `flush=False` - если `True` поток будет сброшен в указанный файл `file` принудительно. Значение по умолчанию `False`

Возвращаемое значение:

- текстовый поток

Описание:

Функцию `print()` выводят объекты в текстовый поток, отделяя их друг от друга `sep` и заканчивая поток `end`. `sep`, `end`, `file` и `flush`, если они заданы, должны быть переданы в качестве аргументов **ключевых слов**.

Ключевые аргументы функции `sep` и `end` должны быть **строками**, они также могут быть `None`, что означает использование значений по умолчанию. Если ничего не передано, `print()` просто напечатает конец строки '\n'.

Переданные объекты в функцию `print()` будут преобразованы в строку по тем же правилам, по которым работает `str()`.

Внимание:

В связи с тем, что объекты преобразуются в строки, функция не может быть использована с бинарными файловыми. Чтобы вести запись в такие файлы используйте `file.write()`.

8. Функция вывода. Форматирование вывода.

Синтаксис:

```
print(*objects, sep='', end='\n', file=sys.stdout, flush=False)
```

Параметры:

- `*objects` - объекты Python
- `sep=''` - строка, разделятель объектов. Значение по умолчанию `None`
- `end='\n'` - строка, которой заканчивается поток. Значение по умолчанию `None`
- `file=sys.stdout` - объект, реализующий метод `writeline(string)`. Значение по умолчанию `None`
- `flush=False` - если `True` поток будет сброшен в указанный файл `file` принудительно. Значение по умолчанию `False`

Возвращаемое значение:

- текстовый поток

Описание:

Функцию `print()` выводят объекты в текстовый поток, отделяя их друг от друга `sep` и заканчивая поток `end`. `sep`, `end`, `file` и `flush`, если они заданы, должны быть переданы в качестве аргументов **ключевых слов**.

Ключевые аргументы функции `sep` и `end` должны быть **строками**, они также могут быть `None`, что означает использование значений по умолчанию. Если ничего не передано, `print()` просто напечатает конец строки '\n'.

Переданные объекты в функцию `print()` будут преобразованы в строку по тем же правилам, по которым работает `str()`.

Внимание:

В связи с тем, что объекты преобразуются в строки, функция не может быть использована с бинарными файловыми. Чтобы вести запись в такие файлы используйте `file.write()`.

Синтаксис:

```
format(value, format_spec)
```

Внимание! Ссылка для тех, кто ищет [метод форматирования строки str.format\(\)](#).

Параметры:

- `value` - форматируемое значение,
- `format_spec` - спецификации формата 'Mini-Language'.

Возвращаемое значение:

- `строка`, отформатированная в соответствии с форматом 'Mini-Language'.

Описание:

Функция `format()` преобразует переданное значение в отформатированную строку, в соответствии с спецификацией формата `Mini-Language`, которое задается в необязательном аргументе `format_spec`.

Интерпретация `format_spec` будет зависеть от типа значения переданного функции аргумента. По умолчанию `format_spec` пустая строка, которая обычно дает тот же эффект, что и вызов функции `str(value)`.

Общий вид инструкции format_spec:

заполнитель | выравнивание | знак | # | размер строки | разделитель десятков | .точность | тип

Спецификация формата Mini-Language:

1. Выравнивание:

- < - Левое выравнивание результата (в пределах доступного пространства);
- > - Выравнивает результат по правому краю (в пределах доступного пространства);
- ^ - Выравнивает результат по центру (в пределах доступного пространства);
- = - Помещает результат в крайнее левое положение;

2. Знаки, только для чисел:

- + - знак плюс;
- - - знак минус только для отрицательных значений;

3. Разделители десятков, только для чисел:

- , - Использовать запятую в качестве разделителя тысяч;
- _ - Использовать символ подчеркивания в качестве разделителя тысяч;

4. Точность:

- .число - количество цифр выводимых после фиксированной точки или количество символов в строке;

```
>>>format(1, 'f')
'1.000000'
>>>format(10, '.2f')
'10.00'
>>>format('format', '.2')
'fo'
```

5. Тип форматируемого объекта:

- **s** - строка, можно не указывать, используется по умолчанию;
 - **b** - двоичный формат;
 - **c** - преобразует целое число в символ Unicode;
 - **d** - десятичный формат;
 - **e** - научный формат, со строчной буквой **e**;
 - **E** - научный формат, с **E** верхним регистром;
 - **f** - формат чисел с плавающей запятой;
 - **F** - формат чисел с плавающей запятой, верхний регистр;
 - **g** - общий формат, нижний регистр;
 - **G** - общий формат, верхний регистр;
 - **o** - Восьмеричный формат;
 - **x** - шестнадцатеричный формат, нижний регистр;
 - **X** - шестнадцатеричный формат, верхний регистр;
 - **n** - формат целых чисел;
 - **%** - Процентный формат. Умножает число на 100 и использует **f** для вывода. В конце ставится **%**;
 - **#** - альтернативный вариант вывода указанного формата, работает с форматами **b**, **o**, **x**.

Пример: `format(1000.5368, '~>+15,.2f')`, где `'~>+15,.2f'` - формат Mini-Language.

- ~ - заполнитель
 - > - выравнивание
 - + - знак
 - 15 - размер итоговой строки в символах
 - , - разделитель десятков
 - .2 - точность, количество выводимых цифр после запятой
 - f - тип,

Пример выведет отформатированную строку: '~~~~~+1,000.54'

```
str.format(*args, **kwargs)
```

Внимание! Ссы...

- ## Параметры:

- `**kwargs` - ключевые аргументы

- ### Возвращаемое значение:

• SCI, ROM

Описание: Метод `str.format()` выполняет операцию форматирования строки `str`. Возвращает копию строки, в которой каждое заменяющее поле заменяется *строковым значением* соответствующего аргумента.

Строки `str` метода `str.format()` содержат "замещающие поля", заключенные в фигурные скобки `{}`. Все, что не содержится в фигурных скобках, считается буквальным текстом, который копируется без изменений в выходные данные. Если вам нужно включить символ скобки в буквальный текст, то это можно сделать путем удвоения фигурных скобки `{}{текст}`.

Каждое замещающее поле {} содержит либо числовой индекс [позиционного аргумента](#), либо имя [ключевого аргумента](#).

Грамматика для поля замены выглядит следующим образом

```
["{" [ field_name ] [ "!" conversion ] [ ":" format_spec ] "}]  
# где:  
# - field_name - это *args и(или) **kwargs  
# - conversion - это 'r' или 's' или 'a'  
# - format spec - это Спецификация формата Mini-Language
```

9. Оператор присваивания. Множественное присваивание.

Оператор присваивания

Оператор присваивания предназначен для связывания имен со значениями и для изменения атрибутов или элементов изменяемых объектов. Оператор присваивания связывает переменную с объектом.

Оператор присваивания обозначается '='.

Обычная (каноническая) форма:

`name = value`

Позиционное присваивание

`name1, name2, ..., nameN = value1, value2, ..., valueN`

Множественное присваивание
переменной

Python позволяет присваивать одно значение нескольким переменным одновременно. Таким образом, вы можете инициализировать несколько переменных, а позже переназначить их. Например, можно задать переменным `x, y` и `z` значение 0. 13 нояб. 2016 г.

```
1 | >>> a = b = c = 1
```

10. Условный оператор. Полные условные операторы. Неполные условные операторы. Тернарный оператор условия. Примеры использования.

Условный оператор или **оператор ветвления** - это **оператор**, конструкция языка программирования, обеспечивающая выполнение **определенной** команды (набора команд) только при условии истинности некоторого логического выражения, либо выполнение одной из нескольких команд (наборов команд) в зависимости от значения некоторого выражения.

```
if выражение1:  
    блок1  
elif выражение2:  
    блок2  
elif выражение3:  
    блок3  
else:  
    блок4
```

Условный оператор. Примеры. Неполный оператор условия

```
m = x  
if x < y:  
    m = y
```

Условный оператор. Примеры. Полный оператор

```
if x < y:  
    print('x меньше y')  
elif x==y:  
    print('x и y равны')  
else:  
    print('x больше y')
```

Ключевое слово `elif` является сокращением от `else if`. Инструкции `elif` и `else` могут отсутствовать и не являются обязательными. Инструкция `elif` может повторяться сколько угодно раз `if/elif/else`. Такая последовательность инструкций `elif` является заменой для конструкции `switch` или `case`, которые есть в других языках программирования.

Конструкция `if` вычисляет и оценивает выражения одно за другим, пока одно из них не окажется истинным, затем выполняется соответствующий блок кода . После выполнения блока кода первого истинного (`True`) выражения, последующие инструкции `elif/else` не вычисляются и не оцениваются, а блоки кода в них не выполняются. Если все выражения ложны (`False`), выполняется блок кода инструкции `else`, если он присутствует. Тернарный оператор – это оператор, который используется для демонстрации какого-то условия, то есть вместо условной конструкции. Он состоит из значений `True` и `False`, а также выражения, которое должно быть вычислено для этих значений.

Синтаксис тернарного оператора:

```
[если истина] if [выражение] else [если ложь]
```

Пример тернарного оператора:

```
1. | x, y = 25, 50  
2. | big = x if x < y else y
```

Тернарный оператор имеет самый низкий приоритет и используется для принятия решения на основе проверки истинности условия.

11. Условные операторы. Множественный выбор. Вложенные операторы условия. Примеры использования.

Условный оператор или **оператор ветвления** - это **оператор**, конструкция языка программирования, обеспечивающая выполнение **определенной** команды (набора команд) только при условии истинности некоторого логического выражения, либо выполнение одной из нескольких команд (наборов команд) в зависимости от значения некоторого выражения.

Про вложенный оператор условия:

Вложенные операторы if позволяют добавить в код второстепенные условия, которые будут проверены, если первичное выражение истинно. То есть, таким образом вы можете поместить одно выражение if-else внутри другого выражения if-else. Синтаксис имеет такой вид:

Вложенные конструкции if:

```
a = 15
b = 3
if a != b
    if a > b:
        print('a больше b')
    else:
        print('a меньше b')
else:
    print('a равно b')
```

Про множественный выбор:

Если требуется написать программу, которая обрабатывает более чем 2 события без использования вложенных операторов if, то для этого понадобится конструкция elif. Она, также как и слово else, является необязательной частью оператора if.

Ключевое слово elif является сокращением от else if. Инструкции elif и else могут отсутствовать и не являются обязательными. Инструкция elif может повторяться сколько угодно раз if/elif/else. Такая последовательность инструкций elif является заменой для конструкции switch или case, которые есть в других языках программирования.



12. Операторы цикла. Цикл с условием. Операторы break и continue. Примеры использования.

Циклы

Цикл – разновидность управляющей конструкции в высокоуровневых языках программирования, предназначенная для организации многократного исполнения набора инструкций.

Основные разновидности:

- бесконечный цикл
- цикл с предусловием
- цикл с постусловием
- цикл со счётчиком

В Python:

- while
- for

Цикл while

while условие:

операторы1

else:

операторы2

Конструкция `while` выполняет код внутри цикла, пока условие остается истинным. Как только условие перестает быть истинным, цикл завершается.

Часть else

Выполняется, если цикл был завершён без прерывания по `break`.

Особенность Python. Позволяет устраниить потребность в дополнительных флагах состояния.

Ключевое слово `break`, как и в C, прерывает выполнение блока `for` или `while` с выходом из него.

Ключевое слово `continue`, заимствован из языка C. Если ход цикла встречает это ключевое слово, то будет пропущен один шаг. Другими словами, цикл продолжится со следующей итерации:

```
count = 3
while count < 7:
    print(count, " < 7")
    count = count + 1
else:
    print (count, " = 7")
```

13. Операторы цикла. Цикл с итератором. Функция `range()`. Примеры использования.

Циклы

Цикл – разновидность управляющей конструкции в высокоуровневых языках программирования, предназначенная для организации многократного выполнения набора инструкций.

Основные разновидности:

- бесконечный цикл
- цикл с предусловием
- цикл с постусловием
- цикл со счётчиком

В Python:

- while
- for

Цикл for

for переменная in объект:

 операторы1

else:

 операторы2

Инструкция `for` в Python не перебирает арифметическую прогрессию чисел, не дает пользователю возможность определять шаг итерации и условие остановки. Вместо этого инструкция `for` в Python перебирает элементы любой последовательности ([список list](#), строку `string`, кортеж `tuple`, словарь `dict` или другого объекта, поддерживающего итерацию) в том порядке, в котором они появляются.

```
>>> words = ['cat', 'window', 'defenestrate']
# проходимся по списку `words`
>>> for w in words:
...     print(w)
...
# cat
# window
# defenestrate
```

Синтаксис:

```
range(stop)
range(start, stop)
range(start, stop, step)
```

Параметры:

- `start` - int, начало последовательности,
- `stop` - int, конец последовательности,
- `step` - int, шаг последовательности.

Возвращаемое значение:

- int, последовательность целых чисел, с заданным шагом.

Описание:

Класс `range()` (диапазон) генерирует арифметическую прогрессию [целых чисел](#), с заданным шагом. По существу это отдельный неизменяемый [тип данных](#) в языке Python. Диапазоны реализуют все [общие операции с последовательностями](#), кроме конкатенации и повторения, поскольку объекты диапазона могут представлять только последовательности, которые следуют строгому шаблону, а повторение и конкатенация обычно нарушают этот шаблон.

```
# отсчет с нуля
>>> for i in range(0, 5):
...     print(i)
...
# 0
# 1
# 2
# 3
# 4
```

14. Изменяемые и неизменяющие типы данных.

Изменяющие и неизменяющие типы в Python

Неизменяющие (immutable):

- bool
- int
- float
- tuple
- str
- frozenset

Изменяющие (mutable):

- list
- set
- dict

Тип данных `bytes` - это **неизменяющие последовательности** отдельных байтов.

`bytearray` объекты являются **изменяющим** аналогом `bytes` объектов.

15. Списки. Основные функции, методы, операторы для работы со списками. Срезы.

Списки представляют собой **изменяющие последовательности**, обычно используемые для хранения коллекций однородных элементов, где степень сходства зависит от приложения.

В Python списки представлены [встроенным классом list\(\)](#), его можно использовать для преобразования итерируемых объектов в [тип list](#).

Функции для работы со списками

- `all()` - возвращает `True`, если все элементы истинны или список пуст
- `enumerate(iterable, start=0)` - возвращает **перечисляемый** объект - кортежи (индекс, значение)
- `len(s)` - количество элементов списка
- `list([iterable])` - создание списка на основе итерируемого объекта, например, `a = list(range(10))`
- `max(iterable)`
- `min(iterable)`
- `print()`
- `reversed(seq)` - возвращает итератор. Не создаёт копию последовательности. `b = list(reversed(a))`. Подходит для `range()` и других объектов, удовлетворяющих требованиям.
- `sorted(iterable, key = None, reverse = False)`
- `sum(iterable)`

Методы списков

- `append(x)` - добавление элемента `x` в конец списка
- `extend(iterable)` - расширение списка с помощью итерируемого объекта
- `insert(i, x)` - вставка `x` в `i`-ю позицию. Если `i` за границами списка, то вставка происходит в конец/начало списка
- `remove(x)` - удаляет первый элемент со значением `x`
- `pop([i])` - удаляет элемент в позиции `i`. Если аргумент не указан, удаляется последний элемент списка
- `clear()` - удаляет все элементы из списка
- `index(x[, start[, end]])` - возвращает индекс (с 0) первого элемента, равного `x`
- `count(x)` - возвращает количество вхождений `x` в список
- `sort(key=None, reverse=False)` - сортировка списка
- `reverse()` - разворачивает список (переставляет элементы в обратном порядке)
- `copy()` - создание "мелкой" копии

Операторы для работы со списками

- `"+"` - конкатенация списков. Аналогично `extend`, но только для списков
- `"*"` - "умножение" списка. `list * число` - увеличить длину списка в `n` раз, скопировав элементы
- `"in"` - принадлежность значения списку: `5 in [3,5,7]`
- `del` - удаление переменной или элемента
- `"=="` - сравнение списков на совпадение элементов с учётом порядка
- `">" ">=" "<" "<="` - сравнение списков с учётом лексикографического порядка элементов
- `for i in list: ...`

Срезы

[start:stop:step] - возвращает элементы списка, начиная с индекса start до stop с шагом step.

a[:] - все элементы списка

a[5:] - с элемента с индексом 5 до конца

a[:2] - элементы 0 и 1

a[::-1] - разворачивание списка

16. Списки. Создание списков. Списковые включения.

Списки представляют собой **изменяемые последовательности**, обычно используемые для хранения коллекций однородных элементов, где степень сходства зависит от приложения.

В Python списки представлены [встроенным классом list\(\)](#), его можно использовать для преобразования итерируемых объектов в [тип list](#).

Создание списков

```
a = [] # пустой список
```

```
a = [0] * 10 # список фиксированного размера, инициализированный  
# начальными значениями
```

Списковое включение ([List comprehensions](#)) – это списки, которые генерируются с **циклом for** внутри.

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

```
a = [i for i in range(10) if i % 2 == 0]
```

Способы ввода списков

```
# 1. быстро, плохо:
```

```
a = list(map(int, input('Введите массив (в одну строку через  
пробел): ').split()))
```

```
# 2. по элементам с указанием размера:
```

```
n = int(input('Введите размер массива: '))
```

```
a = [0] * n
```

```
for i in range(n):
```

```
    a[i] = int(input('Введите {}-й элемент: '.format(i+1)))
```

Способы ввода списков

```
# по элементам без ввода размера:  
a = []  
i = 0  
while True:  
    i += 1  
    el = input('Введите {}-й элемент: '.format(i))  
    if el:  
        a.append(int(el))  
    else:  
        break
```

17. Списки. Основные методы для работы с элементами списка.

Добавление элемента, вставки, удаление, поиск.

Списки представляют собой **изменяемые последовательности**, обычно используемые для хранения коллекций однородных элементов, где степень сходства зависит от приложения.

В Python списки представлены [встроенным классом list\(\)](#), его можно использовать для преобразования итерируемых объектов в [тип list](#).

```
# Проверка наличия значения элемента  
x in sequences  
  
# Проверка отсутствия значения элемента  
x not in sequences
```

Операция позволяет [получить значение элемента по индексу в последовательности](#).

```
sequence[i]
```

- `append(x)` - добавление элемента x в конец списка
 - `extend(iterable)` - расширение списка с помощью итерируемого объекта
 - `insert(i, x)` - вставка x в i-ю позицию. Если i за границами списка, то вставка происходит в конец/начало списка
 - `remove(x)` - удаляет первый элемент со значением x
 - `pop([i])` - удаляет элемент в позиции i. Если аргумент не указан, удаляется последний элемент списка
- **count(x) - возвращает количество вхождений x в список**

Метод позволяет узнать [индекс первого вхождения указанного элемента в последовательность](#). Результатом будет индекс первого вхождения элемента x в последовательность `sequence`.

```
sequence.index(x[, i[, j]])
```

18. Списки. Основные операции со списками. Поиск минимального элемента. Поиск максимального элемента. Нахождение количества элементов. Нахождение суммы и произведения элементов.

Списки представляют собой **изменяемые последовательности**, обычно используемые для хранения коллекций однородных элементов, где степень сходства зависит от приложения.

В Python списки представлены [встроенным классом list\(\)](#), его можно использовать для преобразования итерируемых объектов в [тип list](#).

Функции для работы со списками

- all() - возвращает True, если все элементы истинны или список пуст
- enumerate(iterable, start=0) - возвращает **перечисляемый** объект - кортежи (индекс, значение)
- len(s) - количество элементов списка
- list([iterable]) - создание списка на основе итерируемого объекта, например, a = list(range(10))
- max(iterable)
- min(iterable)
- print()
- reversed(seq) - возвращает итератор. Не создаёт копию последовательности. b = list(reversed(a)). Подходит для range() и других объектов, удовлетворяющих требованиям.
- sorted(iterable, key = None, reverse = False)
- sum(iterable)

Методы списков

- append(x) - добавление элемента x в конец списка
- extend(iterable) - расширение списка с помощью итерируемого объекта
- insert(i, x) - вставка x в i-ю позицию. Если i за границами списка, то вставка происходит в конец/начало списка
- remove(x) - удаляет первый элемент со значением x
- pop([i]) - удаляет элемент в позиции i. Если аргумент не указан, удаляется последний элемент списка
- clear() - удаляет все элементы из списка
- index(x[, start[, end]]) - возвращает индекс (с 0) первого элемента, равного x
- count(x) - возвращает количество вхождений x в список
- sort(key=None, reverse=False) - сортировка списка
- reverse() - разворачивает список (переставляет элементы в обратном порядке)
- copy() - создание "мелкой" копии

Операторы для работы со списками

- "+" - конкатенация списков. Аналогично extend, но только для списков
- "*" - "умножение" списка. list * число - увеличить длину списка в n раз, скопировав элементы
- "in" - принадлежность значения списку: 5 in [3,5,7]
- del - удаление переменной или элемента
- "==" - сравнение списков на совпадение элементов с учётом порядка
- ">" ">=" "<" "<=" - сравнение списков с учётом лексикографического порядка элементов
- for i in list: ...

Срезы

[start:stop:step] - возвращает элементы списка, начиная с индекса start до stop с шагом step.

a[:] - все элементы списка

a[5:] - с элемента с индексом 5 до конца

a[:2] - элементы 0 и 1

a[::-1] - разворачивание списка

- max(iterable)
- min(iterable)

• len(s) - количество элементов списка

- sum(iterable)

Нахождение произведения элементов списка:

```
1 arr = [int(input('Введите элемент списка: ')) for i in range(int(input('Введите длину списка: ')))]
2
3 prod = 1
4 for i in arr:
5     prod *= i
6
7 print(f'Весь список: {arr}')
8 print(f'Сумма элементов списка равна: {sum(arr)}')
9 print(f'Произведение элементов списка: {prod}')
```

19. Списки. Использование срезов при обработке списков.

Списки представляют собой **изменяемые последовательности**, обычно используемые для хранения коллекций однородных элементов, где степень сходства зависит от приложения.

В Python списки представлены **встроенным классом list()**, его можно использовать для преобразования итерируемых объектов в **тип list**.

Срезы удобнее создать при помощи расширенного синтаксиса индексации, квадратных скобок [] с двоеточиями в качестве разделителей внутри. Например: x[start:stop:step] или x[start:stop]

- x[a:b] срез от a до b-1 элементов x[a], x[a+1], ..., x[b-1].
- x[a:b:-1] срез от b до a+1 элементов x[b], x[b-1], ..., x[a+1], то есть меняется порядок элементов.
- x[a:b:k] срез с шагом k: x[a], x[a+k], x[a+2*k],... . Если значение k < 0, то элементы идут в противоположном порядке.
- Каждое из чисел a или b может отсутствовать, что означает "начало последовательности" или "конец последовательности"

Важно: При срезе, первый индекс входит в выборку, а второй нет!

Особенности среза:

- Отрицательные значения старта и стопа означают, что считать надо не с начала, а с конца коллекции.
- Отрицательное значение шага - перебор ведёт в обратном порядке справа налево.
- Если не указан старт `[:stop:step]` - начинаем с самого края коллекции, то есть с первого элемента (включая его), если шаг положительный или с последнего (включая его), если шаг отрицательный (и соответственно перебор идет от конца к началу).
- Если не указан стоп `[start:: step]` - идем до самого края коллекции, то есть до последнего элемента (включая его), если шаг положительный или до первого элемента (включая его), если шаг отрицательный (и соответственно перебор идет от конца к началу).
- `step = 1`, то есть последовательный перебор слева направо указывать не обязательно - это значение шага по умолчанию. В таком случае достаточно указать `[start:stop]`. Можно сделать даже так `[:]` - это значит взять коллекцию целиком

Синтаксис:

```
sequence[i:j]
```

Параметры:

- `sequence` - последовательность. Могут быть `list`, `str`, `tuple`, `set` и т. д.
- `i` и `j` - целые числа

Результат:

- новая последовательности того-же типа

Описание:

Операция позволяет получить срез/часть последовательности от индекса `i` до индекса `j`.

```
>>> x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> x[5:]
```

Синтаксис:

```
sequence[i:j:k]
```

Параметры:

- `sequence` - последовательность. Могут быть `list`, `str`, `tuple`, `set` и т. д.
- `i, j, k` - целые числа

Результат:

- новая последовательность того-же типа

Описание:

Операция позволяет получить срез/часть последовательности от индекса `i` до индекса `j` с шагом `k`.

```
>>> x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> x[::]
```

Синтаксис:

```
sequence[i:j] = iterable
```

Параметры:

- `iterable` - любой объект поддерживающий итерацию
- `sequence` - изменяемая последовательность, `list` или `bytearray`,
- `i` и `j` - целые числа

Результат:

- изменение части элементов последовательности

Описание:

Операция `sequence[i:j] = iterable` позволяет заменить элементы последовательности по указанному срезу содержимым итерации.

```
>>> x = [1, 4, 7, 10, 13, 16, 19]
>>> x[1:4] = [0, 0, 0]
>>> x
# [1, 0, 0, 0, 13, 16, 19]
```

Синтаксис:

```
del sequence[i:j]
```

Параметры:

- `sequence` - изменяемая последовательность, `list` или `bytearray`,
- `i` и `j` - целые числа

Результат:

- удаление части элементов последовательности

Описание:

Операция `del sequence[i:j]` позволяет удалить элементы в последовательности по определенному срезу.

```
>>> x = [1, 4, 7, 10, 13, 16, 19]
>>> del x[1:4]
>>> x
# [1, 13, 16, 19]
```

Синтаксис:

```
sequence[i:j:k] = iterable
```

Параметры:

- `iterable` - любой объект поддерживающий итерацию
- `sequence` - изменяемая последовательность, `list` или `bytearray`,
- `i, j и k` - целые числа

Результат:

- изменение части элементов последовательности

Описание:

Операция `sequence[i:j:k] = iterable` позволяет заменить элементы последовательности по указанному срезу с определенным шагом содержимым итерации.

В результате, указанный срез последовательности `sequence` от индекса `i` до индекса `j` с шагом `k` заменяется содержимым итерации `iterable`.

Важно! Итерация `iterable` должна иметь ту же длину, что и срез последовательности `sequence`, который она заменяет, иначе будет брошено исключение `ValueError`

```
>>> x = [1, 4, 7, 10, 13, 16, 19]
>>> x[1:6:2] = [0, 0, 0]
>>> x
# [1, 0, 7, 0, 13, 0, 19]
```

Синтаксис:

```
del sequence[i:j:k]
```

Параметры:

- `sequence` - изменяемая последовательность, `list` или `bytearray`,
- `i, j и k` - целые числа

Результат:

- удаление части элементов последовательности

Описание:

Операция `del sequence[i:j:k]` позволяет удалить элементы в последовательности по определенному срезу с указанным шагом.

В результате произойдет удаление элементов последовательности `sequence` от индекса `i` до индекса `j` с шагом `k`.

```
>>> x = [1, 4, 7, 10, 13, 16, 19]
>>> del x[1:6:2]
>>> x
# [1, 7, 13, 19]
```

20.Кортежи. Основные функции, методы, операторы для работы с кортежами.

Кортежи – это **неизменяемые последовательности**, обычно используемые для хранения коллекций разнородных данных. Кортежи также используются в тех случаях, когда требуется неизменяемая последовательность однородных данных.

В Python кортежи представлены **классом tuple()**.

Кортежи могут быть созданы несколькими способами:

1. Используя пару скобок для обозначения пустого кортежа: () .

2. Использование запятой для одиночного кортежа: a, или (a,).

3. Разделение элементов запятыми: a, b, c или (a, b, c):

Обратите внимание, что запятая создает кортеж, а не скобки. Скобки необязательны, за исключением случая пустого кортежа, или когда они необходимы, чтобы избежать синтаксической двусмыслистности.

Например:

o f(a, b, c) - вызов функции с тремя аргументами,

o f((a, b, c)) - вызов функции с кортежем в качестве единственного аргумента.

4. Использование встроенного класса tuple():

o tuple() - создаст пустой кортеж,

o tuple(iterable) - преобразует контейнер, поддерживающим итерацию в кортеж.

Операторы для работы с кортежами:

Проверка наличия значения элемента

x in sequences

Проверка отсутствия значения элемента

x not in sequences

- + -- объединение\сложение последовательностей одного типа (создает новый кортеж, старый не изменяется)

Операции эквивалентны

sequence * n

или

n * sequence

Операция позволяет добавить последовательность sequence к самой себе n раз.

(создается новый кортеж, старый не изменяется)

- == -- сравнение кортежей на совпадение с учетом порядка
- >, >=, <, <= -- сравнение кортежей с учетом лексикографического порядка элементов
- for i in tuple: ...

Функции - аналогично спискам (all, enumerate, len, max, min, reversed, sorted, sum)

Методы - index(x) и count(x)

21. Словари. Понятие ключей и значений. Создание словарей.

Основные функции, методы, операторы для работы со словарями.

Словарь `dict` – это **изменяемый контейнерный объект**, который поддерживает поиск произвольных ключей. В языке Python тип данных словарь `dict` представлен [встроенным классом `dict\(\)`](#).

Ключи словаря – это произвольные, неизменяемые (хешируемые) значения. Значения, которые не являются хешируемыми не могут использоваться в качестве ключей. Это такие значения, которые содержат [list списки](#), сами словари `dict` или другие [изменяемые типы](#), которые сравниваются по значению, а не по идентификатору объекта.

Лучше всего рассматривать словарь как набор пар "ключ-значение" с требованием, чтобы ключи были уникальными в пределах одного словаря. Пара фигурных скобок создает пустой словарь: `'{}'`. Размещение разделенного запятыми списка пар `key: value` в фигурных скобках добавляет пары `key: value` в словарь. Так же словари записываются в коде.

Создание: `a, b = dict(), {'odd': 1, 'even': 2}`

`c = dict(short='dict', long='dictionary')`

`d = dict([(1, 1), (2, 4)])`

`e = {a: a ** 2 for a in range(7)}`

`f = dict.fromkeys(['a', 'b'], 100)`

Операторы `del, in, not in, |, |=` (с 3.9).

Методы словарей

- `clear()` - очищает словарь (удаляет все элементы)
- `copy()` - создаёт “мелкую” копию
- `fromkeys(iterable[, value])` - создаёт словарь на основе ключей и значения по умолчанию.
метод класса
- `get(key[, default])` - возвращает значение по ключу либо `default` либо `None`.
- `items()` - возвращает отображение содержимого
- `keys()` - возвращает отображение ключей
- `pop(key[, default])` - удаляет значение из словаря и возвращает его, либо возвращает `default`, либо порождает исключение `KeyError`
- `popitem()` - возвращает последнюю добавленную в словарю пару либо порождает исключение `KeyError`
- `setdefault(key[, default])` - значение по умолчанию для метода `get` на случай отсутствия ключа
- `update([other])` - обновляет значения по другому словарю, кортежу и т.п.
- `values()` - возвращает отображение значений

Функции - те же, но применяются к ключам. Ключи должны быть либо одного типа данных, либо целые.

Получение списка ключей словаря `list(dict)` в Python.

Список ключей словаря можно получить в результате преобразования словаря `dict` в список, используя встроенный класс `list()`. Метод `dict.keys()` возвращает список-представление всех ключей, используемых в словаре `dict`.

Количество элементов в словаре `len(dict)` в Python.

Операции `len(dict)` возвращает количество элементов в словаре `dict`. Функция `len(dictview)` так же возвратит количество элементов в словаре, если `dictview` является представлением словаря `dict`.

Сортировка по ключу:

Так как значение `key` стоит первым, то и ключ для сортировки укажем как `lambda x: x[0]`, где `x` - это кортеж (`key, val`)

```
# исходный словарь
>>> d = {'b': 9, 'a': 3, 'c': 7}
# собственно сама сортировка
>>> sorted_tuple = sorted(d.items(), key=lambda x: x[0])
# получили отсортированный список кортежей,
# отсортированных по первому значению
>>> sorted_tuple
# [('a', 3), ('b', 9), ('c', 7)]
# преобразовываем обратно в словарь
>>> dict(sorted_tuple)
# {'a': 3, 'b': 9, 'c': 7}
```

Сортировка по значению:

Применяя методику сортировки описанную выше, можно легко догадаться как сортировать словарь по значению. Для этого просто укажем в качестве ключа сортировки индекс значения словаря в полученным списке кортежей: `lambda x: x[1]`, где `x` - это кортеж (`key, val`)

```
>>> d = {'b': 9, 'a': 3, 'c': 7}
>>> sorted_tuple = sorted(d.items(), key=lambda x: x[1])
>>> sorted_tuple
# [('a', 3), ('c', 7), ('b', 9)]
# преобразовываем обратно в словарь
>>> dict(sorted_tuple)
# {'a': 3, 'c': 7, 'b': 9}
```

Обратный порядок (реверс) словаря:

```
>>> x = {'five': 5, 'two': 2, 'three': 3, 'one': 1, 'four': 4, 'six': 6}
>>> items = list(x.items())
>>> y = {k: v for k, v in reversed(items)}
>>> y
# {'six': 6, 'four': 4, 'one': 1, 'three': 3, 'two': 2, 'five': 5}
```

22. Множества. Основные функции, методы, операторы для работы с множествами.

Множество - это неупорядоченный набор различных хешированных `hashable` объектов. Обычно множества используются в [тестировании вхождения элемента](#), удаление дубликатов из последовательности и вычисление математических операций, таких как [пересечение](#), [объединение](#), [разность](#) и т. д.

Операторы: `in`, `not in`, `<=`, `<`, `>=`, `>`, `|`, `&`, `-`, `^`.

Функции: `len(s)`

Только для `set`:

`|=`, `&=`, `-=`, `^=`

Метод `sets.isdisjoint()` в Python, отсутствие элементов в множестве.

Метод `sets.isdisjoint()` возвращает `True`, если множество `sets` не имеет общих элементов с итерируемым объектом `other`. Итерируемый объект `other`, это объект поддерживающий итерацию по своим элементам, может быть список, кортеж, другое множество

Метод `sets.issubset()` в Python, вхождение элементов в множество.

Метод `sets.issubset()` позволяет проверить находится ли каждый элемент множества `sets` в последовательности `other`. Метод возвращает `True`, если множество `sets` **является подмножеством** итерируемого объекта `other`, если нет, т

Метод sets.issuperset() в Python, вхождение элементов в множество.

Метод sets.issuperset() позволяет проверить находится ли каждый элемент последовательности other в множестве sets. Метод возвращает True, если множество sets является надмножеством итерируемого объекта other, если нет, то вернет False.

Метод sets.union() в Python, объединение множеств.

Метод sets.union() позволяет объединить множество с двумя или более последовательностями поддерживающих итерирование. Метод возвращает новое множество с элементами из множества sets и элементами вставленными из всех итерируемых объектов *other.

Метод sets.intersection() в Python, пересечение множеств.

Метод sets.intersection() позволяет найти пересечение множества с одной или более последовательностями поддерживающих итерирование. Метод возвращает новое множество с элементами, общими для множества sets и всех итерируемых объектов *other

Метод sets.difference() в Python, разность множеств.

Метод sets.difference() позволяет получить элементы множества, которых нет в одной или более последовательности поддерживающих итерирование. Метод возвращает новое множество с уникальными элементами множества sets, которых нет

Метод sets.symmetric_difference() в Python, симметричная разница

Метод sets.symmetric_difference() позволяет исключить из результата общие элементы для множества и последовательности, операцию еще называют симметричной разницей.

Метод sets.copy() в Python, копия множества.

Метод sets.copy() вернет мелкую копию множества sets. Эта операция поддерживается как неизменяемым frozenset, так изменяемым множеством set.

Только для set:

Метод set.update() в Python, объединение множеств.

Метод set.update() позволяет добавить в множество set, элементы из одной или более последовательности поддерживающих итерирование. Метод возвращает обновленное множество set с добавленными элементами из всех итерируемых объектов *other

Метод set.intersection_update() в Python, пересечение множеств.

Метод set.intersection_update() позволяет сохранить в множестве set только те элементы, которые присутствуют одновременно во всех объектах, участвующих в операции.

Метод set.difference_update() в Python, разница множеств.

Метод set.difference_update() позволяет удалить элементы из множества set, которые присутствуют во всех сравниваемых объектах.

Метод set.symmetric_difference_update() в Python, симметричная разница.

Метод set.symmetric_difference_update() позволяет изменить множество set так, что оно будет содержать уникальные элементы, встречающиеся в самом множестве и последовательности other.

Метод set.add() в Python, добавляет элемент.

Метод set.add() добавляет элемент elem в множество set. Метод игнорирует добавление существующих элементов. В изменяемое множество можно добавлять только неизменяемые объекты.

Метод set.remove() в Python, удаляет элемент по значению.

Метод set.remove() удаляет элемент из множества set с значением elem. Вызывает KeyError, если elem не содержится в множестве.

[Метод set.discard\(\) в Python](#), удаляет элемент если он существует.

Метод `set.discard()` удаляет элемент из множества `set` элемент `elem`, если его значение существует. Метод НЕ вызывает исключений, если значение `elem` отсутствует в множестве.

[Метод set.pop\(\) в Python](#), извлечение и удаление элемента.

Метод `set.pop()` вернет произвольный элемент из множества `set`, а затем удалит его. Метод вызывает исключение `KeyError`, если множество пусто.

[Метод set.clear\(\) в Python](#), очищает множество.

Метод `set.clear()` удаляет все элементы из множества `set`. Метод очищает множество и не возвращает никакого результата.

23. Строки. Основные функции, методы, операторы для работы со строками.
Срезы.

Строчный тип данных

Строка - тип данных, значениями которого является произвольная последовательность символов. Обычно реализуется как массив символов.

Строки в Python - **неизменяемые** объекты

Операторы и функции работы со строками

- +
- *
- `in, not in`

`len(a)` и другие

`max(a), min(a), sorted(a)` – возвращает список отсортированных символов, `str(a)` – преобразует переданное значение в тип `str`, `enumerate (a, start=0)` – возвращает кортеж, содержащий пары ('счётчик', 'элемент') для элементов указанной последовательности.

Основные методы для работы со строками:

- `count(sub[start,end])` - считает неперекрывающиеся вхождения подстроки в строку
- `encode(encoding='utf-8',errors='strict')` - кодирование в заданную кодировку
- `format()`

- `isalnum()` - если все символы буквенно-цифровые и строка не пустая
- `isalpha()` - если все символы - буквенные и строка не пустая
- `isascii()` - если все символы из таблицы ASCII
- `isdecimal()` - если символы цифровые в 10-й с/с и строка не пустая
- `isdigit()` - если символы цифровые в 10-й с/с и строка не пустая
- `isidentifier()` - является корректным идентификатором
- `islower()` - все символы в нижнем регистре и строка не пустая
- `isnumeric()` - все символы являются "числовыми" и строка не пустая
- `isspace()` - все символы "пробельные" и строка не пустая
- `istitle()` - все символы в верхнем регистре и строка не пустая
- `isupper()` - все символы в верхнем регистре и строка не пустая
- `join(iterable)` - конкатенирует строки
- `ljust(width[, fillchar])` - дополняет пробелами справа до заданной ширины
- `lower()` - переводит в нижний регистр
- `lstrip([chars])` - удаляет символы слева
- `replace(old, new[, count])` - заменяет все вхождения подстроки
- `rstrip([chars])` - удаляет завершающие символы
- `split(sep=None, maxsplit=-1)` - возвращает список слов (частей) по разделителю
 - `strip([chars])` - удаляет символы и из начала, и с конца
 - `swapcase()` - меняет регистр
 - **`upper()` - переводит в верхний регистр**

Срезы:

`s[start:stop:step]`

Индексирование используется для получения отдельных символов, взятие среза позволяет получить подстроку.

24. Матрицы. Создание матрицы. Ввод и вывод матрицы. Выполнение операций с элементами матрицы.

Матрицы

В математике - таблица чисел

В программировании - массив массивов (двумерный массив)

a = [[1, 2, 3], [4, 5, 6]]

N x M: N - количество строк, M - количество столбцов

Обращение к элементу: a[i][j] # i - строка, j - столбец

Создание матриц

a = [[0] * m] * n # неправильно!!! создается n ссылок на 1-ю строку

a = [[0] * m for i in range(n)] # правильно

Ввод-вывод матриц

```
n = int(input('Введите количество строк матрицы: '))
m = int(input('Введите количество столбцов матрицы: '))
a = []
for i in range(n):
    a.append([])
    for j in range(m):
        a[i].append(int(input('Введите {}-й элемент {}-й строки:
'.format(i+1, j+1))))
```

```
1 def printMatrix ( matrix ):
2     for i in range ( len(matrix) ):
3         for j in range ( len(matrix[i]) ):
4             print ( "{:4d}".format(matrix[i][j]), end = " " )
5             print ()
```

25. Матрицы. Квадратные матрицы. Обработка верхне- и нижнетреугольных матриц. Работа с диагональными элементами матрицы.

Матрицы

В математике - таблица чисел

В программировании - массив массивов (двумерный массив)

```
a = [[1, 2, 3], [4, 5, 6]]
```

N x M: N - количество строк, M - количество столбцов

Обращение к элементу: a[i][j] # i - строка, j - столбец

Квадратная матрица – двумерный список, в котором количество строк равно количеству столбцов.

Верхнетреугольная матрица:

```
for i in range(len(a)):  
    for j in range(i, len(a)):  
        a[i][j] = k  
        k += 1
```

Верхнетреугольная матрица относительно побочной диагонали:

```
for i in range(len(a)):  
    for j in range(len(a) - i):  
        a[i][j] = k  
        k += 1
```

Нижнетреугольная матрица:

```
for i in range(len(a)):  
    for j in range(i + 1)  
        a[i][j] = k  
        k += 1
```

Нижнетреугольная матрица относительно побочной диагонали:

```
for i in range(len(a)):  
    for j in range(len(a) - 1 - i, len(a)):  
        a[i][j] = k  
        k += 1
```

Работа с диагональными элементами матрицы:

a[i][j] при i = j

или

a[i][i]

26. Отладка программы. Способы отладки.

Отладка программы – этап разработки программы, на котором обнаруживают, локализуют и устраняют ошибки.

При отладке требуется:

- Узнавать текущие значения переменных
- Выяснить по какому пути выполнялась программа

Способы отладки:

- Использование отладочной печати (отладочного вывода)
- Использование отладчика:
 - С заходом в подпрограммы
 - С выполнением подпрограммы как одного оператора
 - До точки останова

27. Подпрограммы. Функции. Создание функции. Аргументы функции. Возвращаемое значение.

Подпрограмма – поименованная или иным образом идентифицированная отдельна функционально независимая часть компьютерной программы.

Подпрограммы

Процедуры

Функции

Параметры подпрограммы - переменные, которые вызывающая программа передаёт подпрограмме

Формальные параметры - те, которые объявлены при описании подпрограммы

Фактические параметры (аргументы) - те, которые передаются в подпрограмму при её вызове

Процедуры – выполняют команды без возврата значений, а функции – возвращают значение.

Функция – исполняемый оператор, принимающий аргументы и возвращающий значение.

Создание функции:

```
def func_name(param):  
    return value
```

Аргументы функции:

Присваивание новых значений аргументам функции не затрагивает вызывающий код.

2. Модификация аргумента внутри функции:

- неизменяемого - создаст копию (не повлияет на вызывающий код)
- изменяемого - повлияет на вызывающий код (изменит значение в нём)

Виды параметров функции:

- Позиционные (*args)
- Именованные (**kwargs)

*args – неопределенное количество позиционных аргументов.

**kwargs – словарь с именованными аргументами.

При определении значения аргумента при описании функции, словарь становится необязательным:

```
def func3(a, b=2): # значения по умолчанию  
    pass
```

```
def func1([posonly1, posonly2, /] pos_or_keyword1, pos_or_keyword2[, *,  
kw_only1, kw_only2]): # позиционные, смешанные, именованные  
    pass
```

только позиционные, разделитель (/), позиционные или
именованные, разделитель (*), только именованные

Возвращаемое значение:

- Возврат значения и передача данных в точку вызова осуществляется через оператор `return`.
- Все строки, идущие после `return` будут проигнорированы.
- При возврате нескольких значений возвращается кортеж.
- При возврате пустого `return` вернется `None`.

28. Функции. Области видимости.

Функция – исполняемый оператор, принимающий аргументы и возвращающий значение.

Пространство имён - множество уникальных идентификаторов (имён).

Область видимости - часть программы, в пределах которой идентификатор остаётся связан с сущностью, которой он был назначен при объявлении.

В Python пространство имён определяется по местоположению присваивания этому имени какого-либо значения.

Основы областей видимости в Python

- имена, присвоенные внутри def, “видны” только в коде внутри этого оператора; ссылаться на них извне функции нельзя;
- имена внутри def не конфликтуют с переменными за пределами def.

Области видимости:

1. Глобальная - если переменная объявлена за пределами всех def, то она является “глобальной” в целом файле.
2. Локальная - переменная, объявленная внутри def, будет локальной в своей функции.
3. Нелокальная - переменная, объявленная внутри def, включающем другие def.
4. Встроенная (built-in).

Изменение областей видимости

Оператор **global** делает имя внутри функции глобальным.

Оператор **nonlocal** делает имя внутри функции нелокальным.

Распознавание имён. Правило LEGB

Поиск имени выполняется последовательно в:

1. local
2. enclosing (объемлющих) функциях
3. global
4. built-in

29. Функции. Завершение работы функции. Рекурсивные функции.
Прямая и косвенная рекурсия.

Функция – исполняемый оператор, принимающий аргументы и возвращающий значение.

Рекурсия. Рекурсивные функции

Рекурсия - вызов подпрограммы из неё же самой:

- непосредственно - простая рекурсия;
- через другие подпрограммы - косвенная рекурсия.

Тело рекурсивной подпрограммы должно иметь не меньше двух альтернативных (условных) ветвей, хотя бы одна из которых должна быть **терминальной**.

Рекурсивная функция – функция, значения которой для данного аргумента вычисляются с помощью значений для предшествующих аргументов.

Виды рекурсивных вызовов

По количеству вызовов:

- линейная - в теле функции присутствует только один вызов самой себя;
- нелинейная - в теле присутствует несколько вызовов.

По месту расположения рекурсивного вызова:

- головная - рекурсивный вызов расположен ближе к началу тела функции;
- хвостовая (концевая) - рекурсивный вызов является последним оператором функции.

Пример рекурсии:

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

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

30.Функции высшего порядка. Замыкания.

Функции высшего порядка

Функция первого порядка - та, которая принимает только значения "простых" (не функциональных) типов и возвращает значения таких же типов в качестве результата.

Функция высшего порядка - та, которая принимает в качестве аргументов или возвращает другие функции.

Пример функции высшего порядка:

```
def f(x):
    return x + 3

def g(function, x):
    return function(x) * function(x)

print(g(f, 7))
# 100
```

Замыкания

Замыкание (closure) в программировании – функция первого класса, в теле которой присутствуют ссылки на переменные, объявленные вне тела этой функции в окружающем коде и не являющиеся её параметрами.

```
def outer():
    x = 1
    def inner():
        print('x in outer function:', x)
    return inner
```

```
def mul(a):
    def helper(b):
        return a * b
    return helper

>>> mul5 = mul(5)
>>> mul5(2)
# 10

>>> mul5(7)
# 35
```

В этом примере `mul()` служит фабрикой для создания и настройки функции `helper()`.

Вызывая `mul5(2)`, мы фактически обращаемся к функции `helper()`, которая находится внутри `mul()`. Переменная `a`, является локальной для `mul()`, и имеет область *enclosing scope* (охватывающая область) в `helper()`. Несмотря на то, что `mul()` завершила свою работу, переменная `a` не уничтожается, т. к. на нее сохраняется ссылка во внутренней функции, которая была возвращена в качестве результата.

31. **lambda**-функции.

Анонимные функции. Оператор **lambda**

Оператор **lambda** создаёт и возвращает объект функции, который будет вызываться позднее, не присваивая ему имени

lambda аргумент1, аргумент2, ...: выражение, использующее аргументы

Используется для сокращения кода в тех местах, где включение оператора **def** не разрешено синтаксисом.

Примеры:

```
# мили в километры
mile = [1.0, 6.5, 17.4, 2.4, 9]
kilometer = list(map(lambda x: x * 1.6, mile))
print (kilometer)
# Выход
[1.6, 10.4, 27.84, 3.84, 14.4]

doit = [(lambda x,y: x+y), (lambda x,y: x-y), (lambda x,y: x*y), (lambda x,y: x/y)]
rez = doit[0](5, 12)
```

32. Аннотации.

Аннотации - способ добавлять произвольные метаданные к аргументам функции и возвращаемому значению

```
def div(a: 'the dividend',
        b: 'the divisor') -> 'the result of dividing a by b':
    """Divide a by b"""
    return a / b
```

Синтаксис создания аннотации:

```
def foo (a: expression, b: expression = 5):
    ...
```

Синтаксис создания аннотации для возвращаемого значения:

```
def sum() -> expression:
    ...
```

Примеры:

```
def greet_all(names: list[str]) -> None:
    for name in names:
        print("Hello", name)
```

33. Функции map, filter, reduce, zip.

Синтаксис:

```
map(function, iterable, ...)
```

Параметры:

- `function` - пользовательская функция, вызывается для каждого элемента,
- `iterable` - последовательность или объект, поддерживающий итерирование.

Возвращаемое значение:

- `map object` - объект итератора.

Описание:

Функция `map()` выполняет пользовательскую функцию `function` для каждого элемента `последовательности`, коллекции или `итератора iterable`. Каждый элемент `iterable` отправляется в функцию `function` в качестве аргумента.

Если в `map()` передаётся несколько `iterable`, то пользовательская функция `function` должна принимать количество аргументов, соответствующее количеству переданных последовательностей, при этом `function` будет применяться к элементам из всех итераций параллельно.

Синтаксис:

```
filter(func, iterable)
```

Параметры:

- `func` - `функция`, которая принимает элемент фильтруемого объекта и должна вернуть `bool` значение,
- `iterable` - `последовательность` или объект поддерживающий `итериование`.

Возвращаемое значение:

- `filter object` - отфильтрованная последовательность.

Описание:

Функция `filter()` отбирает/фильтрует элементы переданного объекта `iterable` при помощи пользовательской функции `func`.

Если функция возвращает `True`, то оставляет элемент, иначе – пропускает.

Синтаксис:

```
from functools import reduce  
  
reduce(function, iterable[, initializer])
```

Параметры:

- `function` - пользовательская функция, принимающая 2 аргумента,
- `iterable` - итерируемая последовательность,
- `initializer` - начальное значение.

Возвращаемое значение:

- требуемое единственное значение.

Применяет функцию `function` к элементам последовательности итерируемого объекта кумулятивно (накопительно): сначала к первым двум элементам (либо к отдельно заданному значению `initializer` и первому элементу), далее к промежуточному результату и следующему элементу. Если `iterable` пуст и не задано `initializer`, то бросается исключение `TypeError`. Если задано `initializer`, но `iterable` пуст – применяется функция к этому значению.

Например `reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])` вычисляет `((((1 + 2) +3) +4) +5)`. Левый аргумент `x` – это накопленное значение, а правый аргумент `y` – это следующий элемент `iterable`.

Функция zip

`zip(*iterables, strict=False)`

Соединяет элементы итерируемых объектов в кортежи.

Параметр `strict` (добавлен в 3.10) приводит к исключению, если длина объектов отличается.

Прекращает выполнение, как только достигнут конец самой короткой последовательности.

34. Декораторы.

Декораторы @

Декоратор – это функция, которая позволяет обернуть другую функцию для расширения её функциональности без непосредственного изменения её кода.

Декоратор принимает функцию в качестве аргумента и возвращает функцию.

Пример:

```
def sample_decorator(func):
    def wrapper():
        print('Я родился...')
        func()
        print('Меня зовут Лунтик!')
    return wrapper

@sample_decorator
def say():
    print('Привет Мир.')

say()
# Я родился...
# Привет Мир.
# Меня зовут Лунтик!
```

35. Знак “_”.

Случаи употребления “_”:

- Хранит значение последнего выражения в интерпретаторе

```
1. print(47)
2. 47
3. print(_)
4. 47
```

- Для игнорирования некоторых значений

```
1. A, B, C, D, _, E, F, G =(10, 20, 30, 40, 50, 60, 70, 80)
```

Здесь,

```
1. A = 10
2. B = 20
3. C = 30
4. D = 40
5. E = 60
6. F = 70
7. G = 80
```

- Для разделения цифр числа

③ Задание специальных значений для имен
переменных или функций:

- name - обозначение приватных переменных, функций

(Python не поддерживает истинной приватности поэтому просто достаточно знать имена необходимых функций, методов).

name - преобразует копии между
когнитивными и стандартными словами (ник и ник)

__name__ - исключение наименования
атрибутов классов (переменные класса)

__name__ - используется для специальных
переменных или методов (__init__ - инициализации
переменных, __len__ - длина атрибута)

36. Модули. Способы подключения.

Модули

Модуль Python - отдельный файл с кодом, который можно повторно использовать в других программах.

Преимущества модулей:

- многократное использование кода
- разбиение пространства имён системы
- реализация разделяемых служб или данных

Способы подключения модулей:

- import tkinter, тогда tkinter.expression
- import tkinter as tk (где tk – псевдоним)
- from tkinter import ttk (импортируется только ttk)
- from tkinter import * (импортирует все переменные)
- import tkinter

37. Модуль math. Основные функции модуля. Примеры использования функций.

Модуль `math` обеспечивает доступ к [математическим функциям](#), определенным стандартом языка С. Данный модуль не поддерживает операции с [комплексными числами](#) и всегда вызывает исключение при их использовании.

За исключением случаев, когда явно указано иное, все возвращаемые значения являются [числами с плавающей запятой](#).

Основные функции модуля:

`math.ceil(x):`

Функция `math.ceil()` возвращает x округленное в большую сторону до ближайшего [целого](#).
`math.comb(n, k)`

Верните количество способов выбрать k элементов из n элементов без повторения и без порядка.

Вычисляется до $n! / (k! * (n - k)!)$ "когда $k \leq n$ " и вычисляется до нуля "когда $k > n$ ".

Число сочетаний.

`math.fabs(x):`

Функция `math.fabs()` возвращает абсолютное значение, модуль числа x . Результат всегда [типа float](#).

`math.factorial(x):`

Функция `math.factorial()` возвращает факториал указанного числа x .

`math.floor(x):`

Функция `math.floor()` возвращает x округленное в меньшую сторону до ближайшего [целого](#).

`math.fmod(x):`

Функция `math.fmod()` возвращает остаток от деления числа x на число y , вычисленный так, как это определено в библиотеке `math` языка С.

`math.fsum(iterable):`

Функция `math.fsum()` возвращает точную сумму значений в итерируемой [последовательности iterable](#). Возвращаемый результат всегда [типа float](#).

`math.gcd(*integers):`

Функция `math.gcd()` возвращает наибольший общий делитель указанных [целочисленных](#) аргументов `*integers`.

`math.lcm(*integers):`

Функция `math.lcm()` возвращает наименьшее общее кратное указанных [целочисленных](#) аргументов `*integers`.

`math.isfinite(x):`

Функция `math.isfinite()` возвращает `False` если x является либо `nan`, либо `inf` или `-inf`, во всех остальных случаях возвращается `True`.

`math.isinf(x):`

Функция `math.isinf()` возвращает `True` в случаях, когда `x` является **отрицательной или положительной бесконечностью**, иначе возвращает `False`.

`math.modf(x):`

Функция `math.modf()` возвращает кортеж из двух чисел (`f, w`) где `f` это дробная, а `w` - целая часть числа `x`. Результат всегда имеет тип `float`.

`math.trunc(x):`

Функция `math.trunc()` отбрасывает дробную часть числа `x`. Результат будет **целым числом**.

`math.perm(n, k=Hem)`

Верните количество способов выбрать `k` элементов из `n` элементов без повторения и с порядком.

Вычисляется до `n! / (n - k)!` "когда `k <= n`" и вычисляется до нуля "когда `k > n`".

Если значение `k` не указано или отсутствует, то значение `k` по умолчанию равно `n`, и функция возвращает `n!` значение .

Число размещений.

`math.exp(x):`

Функция `math.exp()` возвращает `e`, возведенное в степень `x`, где `e=2.718281...` - основание натуральных логарифмов. Функция считает более точно, чем `math.e ** x` или `math.pow(math.e, x)`

`math.expm1(x):`

Функция `math.expm1()` возвращает `e**x - 1`, которое вычисляется значительно точнее, чем `math.exp(x) - 1`, особенно для небольших чисел `x`.

`math.log(x[, base]):`

Функция `math.log()` возвращает логарифм числа `x` по основанию `base`. Если аргумент `base` не указан, то возвращается натуральный логарифм числа `x`.

`math.log10(x):`

Функция `math.log10()` возвращает десятичный логарифм числа `x`, вычисление которого происходит немного точнее, чем `math.log(x, 10)`.

`math.log2(x):`

Функция `math.log2()` возвращает двоичный логарифм числа `x`, вычисление которого происходит немного точнее, чем `math.log(x, 2)`.

`math.pow(x, y):`

Функция `math.pow()` возвращает `x` в степени `y`.

`math.sqrt(x):`

Функция `math.sqrt()` возвращает квадратный корень числа `x`.

Тригонометрические (все в радианах)
 $\cdot \text{acosh}(x), \text{asinh}(x), \text{atanh}(x), \text{asinh}(x), \text{acos}(x), \text{atan}(x)$
 $\cdot \text{acos}(x), \text{asin}(x), \text{atan}(x), \text{sin}(x), \text{cos}(x), \text{tan}(x)$

`math.degrees(x):`

Функция `math.degrees()` преобразует угол `x` из радиан в градусы.

`math.radians(x):`

Функция `math.radians()` преобразует угол `x` из градусов в радианы.

`math.pi`:

Константа `math.pi` возвращает значение математической константы π с точностью, которая зависит от конкретной платформы.

`math.e`:

Константа `math.e` возвращает значение математической константы e с точностью, которая зависит от конкретной платформы.

`math.inf`:

Константа `math.inf` возвращает положительную бесконечность, значение которое является [типовом float](#) и может присутствовать в математических выражениях.

`math.nan`:

Константа `math.nan` возвращает значение "не число" которое является [типовом float](#) и может присутствовать в математических выражениях. Эквивалентно выражению `float('nan')`.

38.Модуль time.

Предоставляет функции для работы со временем.

time() - время эпохи Юникс, Unix-время, время с 01.01.1970 00:00+00

Функция `ctime()` модуля `time` в Python.

Функция `ctime()` модуля `time` преобразует время, выраженное в секундах с начала "эпохи", в строку вида: "Fri Apr 24 15:31:03 2020", представляющую местное время.

Функция `sleep()` модуля `time` в Python.

Функция `sleep()` модуля `time` приостанавливает выполнение вызывающего потока на указанное количество секунд `secs`. Аргумент `secs` может быть числом с плавающей запятой, для указания более точного времени приостановки.

Функция `localtime()` модуля `time` преобразует время, выраженное в секундах с начала эпохи, в именованный кортеж структуры времени `time.struct_time`, в которой учитывается локальное время OS.

Кортеж структуры времени состоит из атрибутов:

№ индекса	Атрибут	Значение
0	<code>tm_year</code>	(пример - 1993)
1	<code>tm_mon</code>	range [1, 12]
2	<code>tm_mday</code>	range [1, 31]
3	<code>tm_hour</code>	range [0, 23]
4	<code>tm_min</code>	range [0, 59]
5	<code>tm_sec</code>	range [0, 61]; смотри примечание 2 в описании функции strftime()
6	<code>tm_wday</code>	range [0, 6], понедельник = 0
7	<code>tm_yday</code>	range [1, 366]
8	<code>tm_isdst</code>	0, 1 or -1; смотрите ниже в описании
N/A	<code>tm_zone</code>	Сокращение названия часового пояса
N/A	<code>tm_gmtoff</code>	Смещение к востоку от UTC в секундах

Функция `gmtime()` модуля `time` преобразует время, выраженное в секундах с начала эпохи, в именованный кортеж структуры времени `time.struct_time` в UTC, в котором флаг `dst` всегда равен нулю.

UTC – всемирное координированное время.

Функция `mktime()` модуля `time` в Python.

Функция `mktime()` модуля `time` обратная функции `time.localtime()`. Преобразует структуру времени `t` в секунды "эпохи" Unix. Возвращает число с плавающей запятой для совместимости с `time.time()`.

Функция `asctime()` модуля `time` в Python.

Функция `asctime()` модуля `time` преобразует кортеж или `struct_time`, представляющие время, возвращаемое `time.gmtime()` или `time.localtime()`, в строку следующего вида: 'Fri Apr 24 15:13:37 2020'.

Функция `strftime()` модуля `time` в Python.

Функция `strftime()` модуля `time` преобразовывает кортеж или структуру времени `time.struct_time`, представляющие время, возвращаемое `time.gmtime()` или `time.localtime()` в строку, указанную аргументом формата `format`.

Решение: (пример) `%m / %d / %Y, %H:%M:%S`

`%Y` - год
`%m` - месяц
`%d` - день

`%H` - час
`%M` - минуты
`%S` - секунды

`%b` - сокр. наим. месяца
`%B` - полное наименование месяца
`%a` - сокр. наим. дня недели
`%A` - полное наименование дня недели

```
>>> import time
>>> time.strftime("%Y-%m-%d", time.localtime())
# '2020-04-24'
```

Функция `strptime()` модуля `time` в Python.

Функция `strptime()` модуля `time` преобразовывает строку `string`, представляющую время в соответствии с форматом `format`. Возвращаемое значение - это `time.struct_time`, возвращаемое `time.gmtime()` или `time.localtime()`.

39. Модуль `random`. Работа со случайными числами.

Псевдослучайные числа

Pseudo-random numbers (PRN)

Вырабатываемая алгоритмически последовательность чисел, обладающих свойствами случайных чисел и используемых взамен последних при решении на ЭВМ ряда классов задач

Генератор псевдослучайных чисел (ГПСЧ, PRNG) – алгоритм, порождающий последовательность чисел, элементы которой почти независимы друг от друга и подчиняются заданному распределению (обычно равномерному).

Недостатки ГПСЧ:

- повторяемость (периодичность) последовательности;
- зависимость значений

Требования к ГПСЧ:

- длинный период;
- эффективность;
- воспроизводимость.

Модуль random

Реализует генерацию псевдослучайных чисел различных распределений.

`random.seed(a=None, version=2):`

Функция `random.seed()` инициализирует генератор или по другому - задает его начальное состояние.

Используется для хранения случайного метода генерации одних и тех же случайных чисел при многократном выполнении кода на одной или разных машинах.

`random.getstate():`

Функция `random.getstate()` возвращает кортеж с параметрами внутреннего состояния генератора, который может быть использован для воссоздания этого состояния.

`random.setstate(state):`

Функция `random.setstate()` задает внутреннее состояние генератора на основе кортежа с его параметрами, который можно получить с помощью функции `random.getstate()`.

`random.randint(a, b):`

Функция `random.randint()` возвращает случайное целое число N из интервала `a <= N <= b`.

```
random.randrange(stop)
random.randrange(start, stop[, step]):
```

Функция `random.randrange()` возвращает случайное целое число из указанного диапазона (`start, stop, step`). Это эквивалентно выбору из `range(start, stop, step)`, но фактически не создает объект диапазона.

```
random.uniform(a, b):
```

Функция `random.uniform()` возвращает случайное число с плавающей запятой `N` из интервала `[a,b]` если `a < b` или из интервала `[b,a]` если `b < a`.

```
random.random():
```

Функция `random.random()` возвращает случайное число с плавающей запятой из интервала `[0.0,1.0)`.

Функция choice() модуля в Python, выбирает случайный элемент.

Функция `random.choice()` модуля `random` возвращает один случайный элемент из непустой последовательности `seq`.

Функция shuffle() модуля random в Python, перемешивает список.

Функция `random.shuffle()` перемешивает изменяемую последовательность `x` случайным образом на месте. Ничего не возвращает.

Функция random.sample() модуля random в Python.

Функция `sample()` модуля `random` возвращает список длины `k` случайных элементов, выбранных из последовательности или множества `population`. Исходная последовательность `population` остается неизменной. Используется для случайной выборки без замены.

```
random.getrandbits(k):
```

Функция `random.getrandbits()` возвращает целое число состоящее из `k` случайных бит.

40. Модуль `copy`. Способы копирования объектов различных типов. “Глубокая” и “мелкая” копии.

Присваивание не копирует объект, а лишь создает ссылку на него.

Модуль `copy` обеспечивает общие операции неглубокого и глубокого копирования.

```
copy.copy(x):
```

Функция `copy.copy(x)` возвращает мелкую копию `x`.

```
copy.deepcopy(x[, memo]):
```

Функция `copy.deepcopy()` возвращает глубокую копию `x`.

Разница между мелким и глубоким копированием актуальна только для составных объектов, содержащих другие объекты, например `списки` или `экземпляры классов`:

- Неглубокая копия создает новый составной объект, а затем (насколько это возможно) вставляет в него ссылки на объекты, найденные в оригинале.
- Глубокая копия создает новый составной объект, а затем рекурсивно вставляет в него копии объектов, найденных в оригинале.

↳ как устроена копирование конк?

В копировании конк, когда создается новый объект, он имеет ссылки на элементы исходного объекта. Если мы попытаемся внести какие-либо изменения во второй копии конкной объект, он не будет отразиться в исходном объекте, т.к. там, где элементы в объектах не делятся ссылкой на другой объект, то есть не делится объект никакой измененности. Иными словами второго объекта не копируется, копируется только ссылка на элементы.

Когда в исходном объекте присутствуют только приватные типы данных, все изменения не видны новому объекту при выполнении в исходном объекте (и наоборот, поскольку эти типы данных неизменяемы и для каждого изменения создается новый объект).

41. Исключения.

Исключения – тип данных, позволяющий классифицировать ошибки и обрабатывать их.

Для обработки исключений используется конструкция try-except.

```
try:  
    ... операторы...  
except ExceptionType [as err]:  
    ... код обработки ошибки...  
except ExceptionType2:  
    ... код обработки ошибки 2...  
except Exception:  
    ... код обработки всех остальных ошибок...  
else:  
    ... при обработке не было ошибок...  
finally:  
    ... завершение обработки ...
```

except Exception: перехватываются все исключения

Создание исключений:

Создание исключений, оператор raise

```
raise Exception('some error...')
```

```
raise ValueError
```

Инструкция `raise` позволяет программисту принудительно вызвать указанное исключение.

```
>>> raise NameError('HiThere')
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
#     NameError: HiThere
```

Цепочки исключений

В процессе обработки одного исключения может произойти (или быть выброшено через `raise`) другое исключение, так, что обработка исключений будет выполняться по цепочке.

Класс Exception. Дочерние классы исключений

```
class MyException(Exception):
```

```
    pass
```

```
class ParametrizedException(Exception):
```

```
    def __init__(self, my_param, msg=None):
```

```
        super().__init__(msg)
```

```
    ...
```

42. Файлы. Программная обработка файлов. Понятие дескриптора.
Виды файлов.

Файл - поименованное место на носителе данных (внешняя память).

В языках программирования обычно применяется концепция, в которой файл является абстракцией, не привязанной к конкретному типу носителя и файловой системе, а работа с файлами осуществляется подобно обработке массива данных.

Однако, в зависимости от технических возможностей, ряд файлов поддерживает **произвольный** доступ, а остальные - только **последовательный**.

Произвольный доступ к файлу – файлы, хранящие информацию в структурированном виде.

Необходимо считывание всего файла с начала.

Последовательный доступ к файлу – файлы, хранящие информацию в неструктурированном виде.

У файлов с последовательным доступом ниже скорость доступа, но они компактнее.

У файлов с произвольным доступом выше скорость доступа, но они проигрывают по компактности.

Файловый дескриптор:

Файловый дескриптор – целое число, которое присваивается системой каждому потоку ввода-вывода при его создании.

- 0 – стандартный ввод (stdin)
- 1 – стандартный вывод (stdout)
- 2 – стандартный вывод ошибок (stderr)

При завершении работы с файлами присвоенный файловый дескриптор освобождается.

Каждый присвоенный дескриптор уникален.

Виды файлов

-
- текстовые файлы
 - структурированные (типовизированные) форматы
 - бинарные файлы

Формат файла определяется его содержимым. Расширение файла обычно соответствует формату файла, но в общем случае никак на него не влияет.

Структурированные форматы – форматы с произвольным доступом, элементы одного типа.

43. Файлы. Режимы доступа к файлам.

Файл – поименованное место на носителе данных.

Открытие файла:

Синтаксис:

```
fp = open(file, mode='r', buffering=-1, encoding=None,
          errors=None, newline=None, closefd=True, opener=None)
```

Параметры:

- `file` - абсолютное или относительное значение пути к файлу или [файловый дескриптор](#) открываемого файла.
- `mode` - необязательно, [строка](#), которая указывает режим, в котором открывается файл. По умолчанию '`r`'.
- `buffering` - необязательно, [целое число](#), используемое для установки политики буферизации.
- `encoding` - необязательно, [кодировка](#), используемая для декодирования или кодирования файла.
- `errors` - необязательно, [строка](#), которая указывает, как должны обрабатываться ошибки кодирования и декодирования. Не используется в -бинарном режиме
- `newline` - необязательно, режим перевода строк. Варианты: `None`, '`\n`', '`\r`' и '`\r\n`'. Следует использовать только для текстовых файлов.
- `closefd` - необязательно, [bool](#), флаг закрытия файлового дескриптора.
- `opener` - необязательно, пользовательский объект, возвращающий открытый дескриптор файла.

Режимы доступа к файлам:

Значения аргумента mode:

- `r` - открывает файл только для чтения,
- `w` - открыт для записи (перед записью файл будет очищен),
- `x` - эксклюзивное создание, бросается исключение `FileExistsError`, если файл уже существует.
- `a` - открыт для добавления в конец файла (на некоторых Unix-системах пишет в конец файла вне зависимости от позиции курсора)
- `+` - символ обновления (чтение + запись).
- `t` - символ текстового режима.
- `b` - символ двоичного режима (для операционных систем, которые различают текстовые и двоичные файлы).

44. Файлы. Текстовые файлы. Основные методы для работы.

Файл – поименованное место на носителе данных.

Текстовый файл — это файл, компонентами которого являются символьные строки переменной длины, заканчивающиеся специальным маркером конца строки.

Основные методы для работы:

[Метод file.close\(\) в Python, закрывает файл.](#)

Метод `file.close()` закрывает открытый файл. Закрытый файл больше не может быть прочитан или записан. Любая операция, которая требует, чтобы файл был открыт, вызовет исключение `ValueError` после того, как файл был закрыт.

Метод file.read() в Python, читает весь файл или кусками.

Метод файла file.read() считывает из файла не более size байтов. Если достигается конец файла EOF до получения указанного размера size байтов, тогда метод считает только доступные байты.

Метод file.readline() в Python, читает файл построчно.

Метод файла file.readline() читает одну целую строку из файла. Конечный символ новой строки \n сохраняется в строке.

Метод file.readlines() в Python, получает список строк файла.

Метод файла file.readlines() читает файловый объект построчно, пока не достигнет конца файла EOF, и возвращает список, содержащий строки файла.

Метод file.write() в Python, пишет данные в файл.

Метод файла file.write() записывает строку str в файл file. Метод возвращает целое число - количество записанных байт.

Метод file.writelines() в Python, пишет список строк в файл.

Метод файла file.writelines() записывает последовательность строк в файл file.

Обратите внимание, что метод file.writelines() не добавляет разделители строк автоматически. Если они требуются, то добавляйте их вручную.

Метод file.truncate() в Python, усекает размер файла.

Метод файла file.truncate() усекает размер файла. Если указан необязательный аргумент size, файл усекается до этого (максимально) размера.

Метод файла `file.seek()` устанавливает текущую позицию в байтах `offset` для указателя чтения/записи в файле `file`.

Аргумент `whence` является необязательным и по умолчанию равен `0`. Может принимать другие значения:

- `0` - означает, что нужно сместить указатель на `offset` относительно начала файла.
- `1` - означает, что нужно сместить указатель на `offset` относительно относительно текущей позиции.
- `2` - означает, что нужно сместить указатель на `offset` относительно конца файла.

Обратите внимание:

- Если файл открыт для добавления с помощью '`a`' или '`a+`', все операции `file.seek()` будут отменены при следующей записи.
- Если файл открыт только для записи в режиме добавления с использованием '`a`', Этот метод по существу используется, но он остается полезным для файлов, открытых в режиме добавления с включенным чтением - режим '`a+`'.
- Если файл открыт в текстовом режиме с помощью '`t`', то допустимы только смещения, возвращаемые функцией `file.tell()`. Использование других смещений вызывает неопределенное поведение.

Обратите внимание, что не все объекты файлов доступны для метода `file.seek()`. В текстовых файлах разрешены только запросы относительно начала файла, исключение составляет смещение указателя до самого конца файла с помощью `txt_file.seek(0, 2)`.

Метод file.tell() в Python, позиция указателя в файле.

Метод файла `file.tell()` возвращает текущую позицию указателя чтения/записи в файле в байтах.

45.Файлы. Текстовые файлы. Чтение файла. Запись в файл. Поиск в файле.

Файл – поименованное место на носителе данных.

Текстовый файл — это файл, компонентами которого являются символьные строки переменной длины, заканчивающиеся специальным маркером конца строки.

Способы чтения файла	Способы записи в файл
<pre>f = open('in.txt') # 1 способ чтения content = f.read() # 2 способ чтения lines_list = f.readlines() # 3 способ чтения for i in range(5): string = f.readline() # 4 способ чтения for s in f: print(s) f.close()</pre>	<pre>f = open('in.txt', 'w') # 1 способ записи f.write('abcd') # 2 способ записи f.writelines(['1', 'abcd']) # 3 способ записи print('def', file=f) f.close() ## нет символа перевода строки</pre>

46. Файлы. Текстовые файлы. Итерационное чтение содержимого файла.

Файл – поименованное место на носителе данных.

Текстовый файл — это файл, компонентами которого являются символьные строки переменной длины, заканчивающиеся специальным маркером конца строки.

Итерационное чтение – чтение файла построчно, без хранения его содержимого в памяти.

```
# first:
string = f.readline()
while string != '':
    # second:
    while True:
        string = f.readline()
        if not string:
            break
    # third:
    for s in f:
        print(s)
```

47. Файлы. Бинарные файлы. Основные методы. Сериализация данных.

Файл – поименованное место на носителе данных.

Бинарные файлы — это файлы, в которых информация хранится в виде набора байтов.

Основные методы:

Синтаксис:

```
str.encode(encoding="utf-8", errors="strict")
```

Параметры:

- `encoding` - `str`, используемая кодировка строки, по умолчанию `utf-8`
- `errors` - обработчик ошибок кодировки.

Возвращаемое значение:

- `bytes`, байтовая версия строки.

Описание:

Метод `str.encode()` вернет закодированную версию строки `str` как объект байтов. Другими словами кодирует текстовую строку `str` в строку байтов, используя указанную кодировку `encoding`.

Аргумент `encoding` по умолчанию используется '`utf-8`'. Для получения списка всех схем кодированиясмотрите [Стандартные кодировки](#).

```
str(object=' ')
str(object=b'', encoding='utf-8', errors='strict')
```

Если задан хотя бы один из аргументов `encoding` или `errors`, то `object` должен быть байтоподобным, например `bytes` или `bytearray`. В этом случае `str(bytes, encoding, errors)` эквивалентен вызову `bytes.decode(encoding, errors)`.

Синтаксис:

```
bytes.decode(encoding='utf-8', errors='strict')
bytearray.decode(encoding='utf-8', errors='strict')
```

Параметры:

- `bytes` и `bytearray` - соответствующие типы `bytes` или `bytearray`,
- `encoding='utf-8'` - кодировка,
- `errors='strict'` - обработчик ошибок кодировки.

Возвращаемое значение:

- строку `str`

Описание:

Метод `decode()` возвращает строку, декодированную из заданных байтов. Кодировка `encoding` по умолчанию - "utf-8". Обработчик ошибок `errors` может быть задан для другой схемы обработки ошибок. Значение по умолчанию для `errors` - 'strict', что означает, что ошибки кодирования вызывают исключение `UnicodeError`. Другими возможными значениями являются 'ignore', 'replace' и любое другое имя, зарегистрированное с помощью `codecs.register_error()`, см. раздел [обрабатчики ошибок](#). Список возможных кодировок см. [стандартные кодировки](#).

Примечание: передача аргумента `encoding` в строку `str` позволяет декодировать любой байт-подобный объект напрямую, без необходимости создавать временный `bytes` объект или объект `bytearray`.

Примеры использования:

Если вы хотите кодировать и декодировать текст, то именно для этого существуют методы `str.encode()` и `bytes.decode()`:

Синтаксис:

```
bytes(source, encoding, error)
```

Параметры:

Все параметры являются обязательными.

- `source` - объект Python,
- `encoding` - кодировка источника, если `source` - это строка,
- `error` - обработчик ошибок. Вызывается в случае неправильной кодировки.

Возвращаемое значение:

- `bytes` - байтовый объект (строка байтов), **неизменяемая последовательность**.

Описание:

Класс `bytes()` возвращает байтовый объект `bytes`, который является неизменяемой последовательностью целых чисел в диапазоне от `0 <= x <256`.

Синтаксис:

```
bytearray(source, encoding, error)
```

Параметры:

Все параметры являются обязательными.

- `source` - объект Python,
- `encoding` - кодировка источника, если `source` - это строка,
- `error` - обработчик ошибок. Вызывается в случае неправильной кодировки.

Возвращаемое значение:

- `bytearray` - массив байтов, **изменяемая последовательность**.

Описание:

Класс `bytearray()` возвращает массив байтов `bytearray`, который является изменяемой последовательностью целых чисел в диапазоне от `0 <= x <256`.

Сериализация данных – процесс перевода какой-либо структуры данных в последовательность битов.

Модули marshal, pickle, shelve

Модули предназначены для сериализации объектов Python в бинарные структуры с целью сохранения на диск либо передачи на другой компьютер.

Синтаксис:

```
import pickle

pickle.dump(obj, file, protocol=None, *, \
            fix_imports=True, buffer_callback=None)

pickle.dumps(obj, protocol=None, *, \
            fix_imports=True, buffer_callback=None)
```

Параметры:

- `obj` - объект Python, подлежащий сериализации,
- `file` - [файловый объект](#),
- `protocol=None` - протокол сериализации
- `fix_imports=True` - сопоставление данных Python2 и Python3,
- `buffer_callback=None` - сериализация буфера в файл как часть потока `pickle`.

Возвращаемое значение:

- `pickle.dumps()` - [строка байтов](#).

Описание:

Функция `pickle.dump()` модуля `pickle` записывает сериализованное представление объекта `obj` в открытый [файловый объект](#). Это эквивалентно вызову `pickle.Pickler(file, protocol).dump(obj)`.

Функция `pickle.dumps()` возвращает выбранное представление объекта `obj` как объекта байтов вместо записи его в файл.

Синтаксис:

```
import pickle

pickle.load(file, *, fix_imports=True, encoding="ASCII",
            errors="strict", buffers=None)

pickle.loads(data, *, fix_imports=True, encoding="ASCII",
            errors="strict", buffers=None)
```

Параметры:

- `file` - [файловый объект](#),
- `data` - [строка байтов](#) с упакованными данными,
- `fix_imports=True` - сопоставление данных Python2 и Python3,
- `encoding="ASCII"` - кодировка для чтения данных, генерируемых Python2,
- `errors="strict"` - [обработчик ошибок декодирования](#),
- `buffers=None` - аргумент `buffer_callback` указанный при создании объекта `pickle.Unpickler()`.

Возвращаемое значение:

- иерархия восстановленных объектов.

Описание:

Функция `pickle.load()` модуля `pickle` читает упакованное представление объекта из открытого [файлового объекта](#) `file` и возвращает указанную в нем восстановленную иерархию объектов. Данная функция эквивалентна вызову `Unpickler(file).load()`.

Функция `pickle.loads()` возвращает восстановленную иерархию объектов из [строкового](#) представления данных объекта `data`. Данные `data` должны быть [байтоподобным](#) объектом.

48. Файлы. Оператор `with`. Исключения.

Файл – поименованное место на носителе данных.

Оператор `with` не требует закрытия файла, так как выполняет это автоматически.

`with open(...) as f1, open(...) as f2:`

операторы

Исключения при работе с файлами:

Ошибки возможны:

- При открытии файла (`FileNotFoundException`, `UnicodeDecodeError`)
- При записи (`PermissionError`)
- При других операциях

49. Типы данных `bytes` и `bytearray`. Байтовые строки. Конвертация различных типов в байтовые строки и обратно.

`bytes` и `bytearray` - классы для представления бинарных данных, “байтовые строки”.

Набор операторов и методов похож на аналогичный у обычных строк.

```
b'bytes'  
'Строка'.encode('utf-8')  
bytes('string', encoding='utf-8')  
bytes([1,2,3,4])  
bytearray(b'string')  
  
bytes - неизменяемый, bytearray - изменяемый
```

Синтаксис:

```
bytes(source, encoding, error)
```

Параметры:

Все параметры являются обязательными.

- `source` - объект Python,
- `encoding` - кодировка источника, если `source` - это строка,
- `error` - обработчик ошибок. Вызывается в случае неправильной кодировки.

Возвращаемое значение:

- `bytes` - байтовый объект (строка байтов), **неизменяемая последовательность**.

Описание:

Класс `bytes()` возвращает байтовый объект `bytes`, который является неизменяемой последовательностью целых чисел в диапазоне от `0 <= x <256`.

Конвертация из байтовой строки:

Синтаксис:

```
bytes.decode(encoding='utf-8', errors='strict')  
bytearray.decode(encoding='utf-8', errors='strict')
```

Параметры:

- `bytes` и `bytearray` - соответствующие типы `bytes` или `bytearray`,
- `encoding='utf-8'` - кодировка,
- `errors='strict'` - обработчик ошибок кодировки.

Возвращаемое значение:

- строку `str`

Описание:

Метод `decode()` возвращает строку, декодированную из заданных байтов. Кодировка `encoding` по умолчанию - "utf-8". Обработчик ошибок `errors` может быть задан для другой схемы обработки ошибок. Значение по умолчанию для `errors` - 'strict', что означает, что ошибки кодирования вызывают исключение `UnicodeError`. Другими возможными значениями являются 'ignore', 'replace' и любое другое имя, зарегистрированное с помощью `codecs.register_error()`, см. раздел [обработчики ошибок](#). Список возможных кодировок см. [стандартные кодировки](#).

Примечание: передача аргумента `encoding` в строку `str` позволяет декодировать любой байт-подобный объект напрямую, без необходимости создавать временный `bytes` объект или объект `bytearray`.

Примеры использования:

Если вы хотите кодировать и декодировать текст, то именно для этого существуют методы `str.encode()` и `bytes.decode()`:

50. Модуль `struct`.

Формирует упакованные двоичные структуры данных из переменных базовых типов данных и распаковывает их обратно.

`struct.pack(format, v1, v2, ...):`

Функция `struct.pack()` возвращает объект [байтов](#), содержащий значения `v1`, `v2`, ..., упакованные в соответствии с форматом `format` [строки формата](#). Аргументы должны точно соответствовать значениям, требуемым форматом `format`.

`struct.pack_into(format, buffer, offset, v1, v2, ...):`

Функция `struct.pack_into()` упаковывает значения `v1`, `v2`, ... в соответствии с форматом `format` [строки формата](#) и запишет упакованные [байты](#) в буфер `buffer` предназначенный для записи, начиная с позиции `offset`.

Обратите внимание, что смещение `offset` является обязательным аргументом.

`struct.unpack(format, buffer):`

Функция `struct.unpack()` распаковывает буфер `buffer`, предположительно упакованного [функцией struct.pack\(format, ...\)](#) в соответствии с форматом [строки формата](#).

Результатом работы функции `struct.unpack()` является [кортеж](#), даже если он содержит ровно один элемент.

Размер буфера в байтах должен соответствовать размеру, требуемому форматом, как отражено в `struct.calcsize()`.

`struct.unpack_from(format, buffer, offset=0):`

Функция `struct.unpack_from()` распаковывает буфер `buffer`, начиная с позиции смещения `offset`, в соответствии с форматом `format` [строки формата](#).

Результатом работы функции `struct.unpack()` является [кортеж](#), даже если он содержит ровно один элемент.

Размер буфера в байтах, начиная со смещения позиции, должен быть не меньше размера, требуемого форматом, как отражено в `struct.calcsize()`.

`struct.iter_unpack(format, buffer):`

Функция `struct.iter_unpack()` лениво распаковывает буфер `buffer` в соответствии с форматом [строки формата](#).

Эта функция возвращает [итератор](#), который будет читать куски одинакового размера из буфера, пока все его содержимое не будет использовано. Каждая итерация дает кортеж, как указано в строке формата `format`.

Размер буфера в байтах должен быть кратным размеру, требуемому форматом, как отражено в `struct.calcsize()`.

`struct.calcsize(format):`

Функция `struct.calcsize()` возвращает размер структуры и следовательно байтового объекта, созданного функцией `struct.pack(format, ...)`, соответствующий формату `format` [строки формата](#).

Форматы:

Символ	Порядок байт	Размер	Выравнивание
@	Естественный (зависит от хост-системы)	Естественный	Естественное
=	Естественной	Стандартный	Отсутствует
<	little-endian	Стандартный	Отсутствует
>	big-endian	Стандартный	Отсутствует
!	сетевой (аналог big-endian)	Стандартный	Отсутствует

Выражение:

@ - нейтральное, но умножение

= - коридок-нейтральный, перед стандартн.

<- от логарифма к степени

> от степени к логарифму ; ! - симметрическое >

Форматы:

c - char

i - int

l - long (integer)

g - long long (integer)

f - float

s - char[] (bytes)

51. Модуль `os`. Основные функции.

Предоставляет функции для работы с операционной системой.

Функции:

Смена рабочей директории из кода.

Синтаксис:

```
import os  
os.chdir(path)
```

Функция `getcwd()` модуля `os` в Python.

Функция `getcwd()` вернет строку, а функция `getcwdb()` вернет строку байтов представляющую текущий рабочий каталог.

Получить список файлов в директории/каталоге.

Синтаксис:

```
import os  
  
os.listdir(path='.' )
```

Параметры:

- `path` - путь в виде [строки](#) или [дескриптор каталога](#).

Создать каталог и установить режим доступа к нему.

Синтаксис:

```
import os  
  
os.mkdir(path, mode=0o777, *, dir_fd=None)
```

Параметры:

- `path` - имя каталога ([стр](#) путь в файловой системе),
- `mode=0o777` - [режимом доступа к каталогу](#),
- `dir_fd=None` - [дескриптор каталога](#)

Удалить файл из кода.

Синтаксис:

```
import os  
  
os.remove(path, *, dir_fd=None)  
os.unlink(path, *, dir_fd=None)
```

Параметры:

- `path` - [стр](#), путь к файлу,
- `dir_fd=None` - [int](#), дескриптор каталога.

Удалить пустой каталог в файловой системе.

Синтаксис:

```
import os  
  
os.rmdir(path, *, dir_fd=None)
```

Параметры:

- `path` - [стр](#), путь в файловой системе до каталога,
- `dir_fd=None` - [int](#), дескриптор каталога,

```
import os  
  
os.removedirs(path)
```

Удаляет пустую директорию, затем пытается удалить родительские директории, пока они пусты.

Переименовать файл или пустой каталог.

Синтаксис:

```
import os

os.rename(src, dst, *, src_dir_fd=None, dst_dir_fd=None)
```

Параметры:

- `src` - `str`, исходное имя файла или каталога,
- `dst` - `str`, новое имя файла или каталога,
- `src_dir_fd=None` - `int`, исходный дескриптор каталога,
- `dst_dir_fd=None` - `int`, новый дескриптор каталога,

```
import os

os.replace(src, dst, *, src_dir_fd=None, dst_dir_fd=None)
```

Функция `replace()` модуля `os` переименовывает файл или пустой каталог с исходным именем `src` в `dst`.

```
import os

os.truncate(path, length)
```

Функция `truncate()` модуля `os` обрезает файл, соответствующий пути `path`, так, чтобы он имел длину не более `length` байтов.

```
import os

stat_result = os.stat(path, *, dir_fd=None, follow_symlinks=True)
```

Возвращаемое значение:

- объект `os.stat_result`

Статистическая информация файла.

```
import os

os.system(command)
```

Функция `system()` модуля `os` выполняет команду `command` в подоболочке (`subshell`).

Возвращаемое значение:

- индикация состояния выхода.

52. Генераторы.

Генератор – функция, которая возвращает объект итератора. Возвращает по одному значению, и при этом замораживает свое выполнение, и при новом вызове этой функции она будет выполняться с того места, на котором она остановилась.

Итератор – объект, который поддерживает функцию `next()` для последовательной генерации значений. Также это объект, порождаемый генератором.

Ключевое слово **yield** (используется в функциях)

Генераторные выражения – выражения, возвращающие итератор генератора.

- генераторные выражения: (i^{**2} for i in range(10))

Также генератор – коллекция, которую можно про итерировать всего один раз.

Преимущества генератора:

Элементы генератора не хранятся все вместе, а формируются на лету. Все элементы генератора не хранятся в памяти.

53. Модуль [numpy](#). Обработка массивов с использованием данного модуля.

numpy - библиотека языка Python ориентирована на работу с многомерными массивами и матрицами, с полиномами и с другими объектами.

Основным объектом numpy является однородный многомерный массив элементов одного типа (называется `numpy.ndarray`).

Некоторые атрибуты `ndarray`

- `ndarray.ndim` - число измерений(принято называть осями) массива.
- `ndarray.shape` - кортеж натуральных чисел, показывающий длину массива по каждой оси. Число элементов кортежа `shape` равно `ndim`.
- `ndarray.dtype` - тип элементов массива. numpy имеет собственные типы данных, например: `float32`, `complex64` и т. д.
- `ndarray.itemsize` - размер каждого элемента в байтах.

```
import numpy  
import numpy as np  
from numpy import *
```

+ Создание массивов

.array(seq, dtype=...) - создание массива, где

seq - итерабл (когда обь → [...], где обь → [...], [...])

а. т.д.), dtype - преобразование типа

.tolist() - переводит из массива в список

.arange([start,] stop, [step,] dtype=None) - создает массив (numpy.ndarray) из н., равномерно распределенных в заданном интервале

. start - начало интервала (по умолчанию = 0)

. stop - конец интервала

. step - шаг (по умолчанию = 1)

. dtype - тип данных

. linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0) - создает массив N н.р., равномерно распределенных в заданном интервале

endpoint - True ⇒ stop включается, иначе - нет

retstep - True ⇒ создает кортеж (array(...), step)

↳ Матрицы симметрического буга

- ① Тиреал (на основе gen в метод, значение которого задается от coefficient) например). `empty((row, column))`
- ② Единичная → `.identity(size)` или `.eye(size)`
- ③ Нулевая → `.zeros((row, column), type=float?)`
- ④ Ч н единичн → `.ones((row, column), type=..)`
- ⑤ Заполненная эл-ами → `.full((row, column), fill)`

↳ Изменение массивов (размерности)

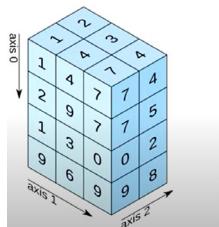
• `.reshape('array', (row, column))` — изменение формы массива
объекта как-то з-о б генерно связанный с первоначальными
ко-вами з-о б

• `.resize('array', (row, column))` — изменение
формы. При несоблюдении к-ва з-о б генерируется
исключение с надписью з-о б последовательности
и `.copy('array')` — копия конца (не из метода `copy`)

`d2.shape=(row, column, page)`

Формирование трехмерного массива.

3D array



• \top - транспонирование

(column)
axis=2

• $\min()$, $\max()$, $\sum()$ - значение для всей матрицы

{
• $\min(\text{axis}='\text{num}')$ } - возвращают список из значений
• $\max(\text{axis}='\text{num}')$ } наибольших по модулю чисел

54. Модуль numpy. Работа с числами и вычислениями.

54. Модуль numpy. Работа с числами и вычислениями

• Сложение $\rightarrow a+b$ - только одинакового порядка

• Вычитание $\rightarrow a-b$

• Умножение членов $\rightarrow a*b$

• Возведение в степень $\rightarrow a**b$

• Умножение на число $\rightarrow a * \text{const}$

• Матричное умножение (2×2) $\rightarrow \text{dot}(a, b)$

матрица
 2×2

. Матричное умножение (2×3) $\rightarrow .dot(a, x)$
 $(2 \times 2) \quad (2 \times 3)$

. С делением на ноль: $\rightarrow a / a_0$

• бывает неприменимо, because $a_0 = \text{inf}$ ^{* присутствует 0-й эл. в 1-т}

. Остаток от деления $\rightarrow a \% a_0$. Аналогично, если делить на "0", бывает неприменимо, остаток - 0.

. Суммирование (глобальный метод) $\rightarrow .add(a, b)$

. Вычитание $\rightarrow .subtract(a, b)$

. Умножение $\rightarrow .multiply(a, b)$

. Деление $\rightarrow .divide(a, b)$

. Смена знака $\rightarrow .negative(a)$

. Транспонирование $\rightarrow .transpose(a)$

↳ Сортировка

. $.sort(\text{axis}=-1, \text{kind}='quicksort', \text{order}=\text{None})$

axis - ось сортировки (нумер. - начиная с 0)

kind - тип сортировки.
'quicksort' - быстрая
'mergesort' - слиянием
'heapsort' - выбором через кучу
'timsort' - слиянием + бинар.

* можно сортировать по любому параметру

Пример.

```
dtype = [('weight',int), ('height',float), ('age', int)]
v = [(70, 1.75, 15), (65, 1.8, 19), (45, 1.85, 16)]
c = np.array(v, dtype=dtype)
c.sort(axis=0, order = ['weight'])
print(c)
```

сортируем столбцы (axis=0) по массе
* Можно установить несколько параметров.

55. Модуль `matplotlib`. Построение графиков в декартовой системе координат. Управление областью рисования.

Matplotlib

Пакет Matplotlib является основным для визуализации расчетных данных.

Для рисования графика используем модуль `pyplot` из пакета.

52. Модуль `matplotlib`. Построение графиков в декартовой системе координат. Управление областью рисования

↳ `Matplotlib` – пакет для визуализации расчетных данных. Для рисования графика используется модуль `pyplot` из пакета

↳ Функция `plot`

`plot(x)` – график зависимости одного числа в зависимости от `x`.

`plot(x, y)` – график, где `x` – значение аргумента, `y` – значение функции (все массивы)

`plot(x, y, s)` – график с возможностью изменения цвета, толщины, способ отображения и т.д.

Tun	Tun	Tun	Убс
шарик	шарик	шарик	шарик
Направлена! '-'	Точка (пункт) но направлена!)	'.' '0'	Синий '8'
Пунктова! '--'	Звездочка '*' 'p'		Черный 'k'
Линия! ':'	Стрелка '+'		Белый 'w'
Линия! '--'	Вертикальные линии ' '		Голубой 'c'
Линия! '--'	Крестик 'x'		Красный 'r'
	Очертан:		Желтый 'j'
Треуг. вправо <			Зеленый 'g'
Треуг. влево >			Малинов.
квадрат 's'			'm'
Шестиугольник 'h'			
Ромб 'd'			
Угол лев 'l'			

Также задается через аргументы:

`linewidth (lw)` - толщина линий

`markerSize (ms)` - размер маркеров (точки)

`to y=markerSize = 1`

↳ Значок, название оси

`title (fontSize, ha, va)` - название графика

`fontSize : 'large', 'medium', 'small'`

`horizontalAlignment (ha) : 'center', 'left', 'right'`

`verticalAlignment (va) : 'top', 'baseline', 'bottom'`

Analogично для осей

. xlabel (...) * Использование окружается знаком '*'
. ylabel (...) где это, чтобы не воспринималось
как предыдущий символ

↳ Установка границы на оси.

. axis ([start_x, stop_x, start_y, stop_y]) - вектор

установки границ для оси Ox : startx, stop_x; Oy : starty, stop_y

. axis('off') - удаляет ось

. axis('equal') - выравнивает границы по оси

(происходит не всегда, а некоторое выравнивание)

↳ Для отображения координатной сетки

. grid (*True*)

↳ Контроль шага координатной сетки

. xticks (seq[, name-seq]) seq - принимает последова-
тельность из чисел

. yticks (seq[, name-seq]) name-seq - последовательность
из строк (и не только). Переиспользовать название
шага заданный name-seq. \Rightarrow если seq и name-seq равны

↳ Разбиение графического окна

. subplot (m, n, p) map(m, n, p) - 3 цифры,

. subplot (mnp) краинвает разбивку графического

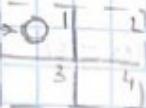
окна на несколько подокон, где

m - количество частей по горизонтали

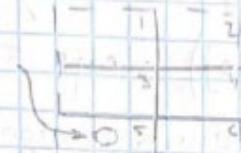
n - кол-во частей по вертикали

p - номер подокна

пример: `subplot(221)`



`subplot(325)`



`subplot(1,1,1)` // `subplot(111)` - выделяют главное окно в именное состояние

• `subplot(...)` вызывается перед обращением к функциям построения (`-plot/-show//...`)

↳ функция `.show()` выводит на экран (вызывается в конце)

↳ обращение к картинке

`legend(loc=0)` - создание "легенды"

Для этого понадобится задать аргумент `label` в

виде строки внутри функции `plot(..., label='sin(x)')`

аргумент `loc` может принимать следующие значения:

'best'	0	'lower right'	4	'lower center'	8
'upper right'	1	'right'	5	'upper center'	9
'upper left'	2	'center left'	6	'center'	10
'lower left'	3	'center right'	7		

56. Модуль `matplotlib`. Построение гистограмм и круговых диаграмм.

Гистограмма

Гистограмма используется для изображения зависимости частоты попадания элементов в соответствующий интервал группировки

*.hist([seq], n=10) - строит гистограмму по
основании значений последовательности seq и
разбивает на интервалы n (по умолч. 10)*

```
Import matplotlib.pyplot as plt
import numpy as np
x = np.random.normal(0, 3, 1000) # np.random.randn(1000)
plt.hist(x, 25) # по умолчанию 10
plt.show()
```

numPy.random.normal(loc = 0.0, scale = 1.0, size = None) –
массив заполненных случайными значениями нормального
распределения/гауссова распределения.
loc определяет среднее значение распределения, **scale** определяет
стандартное отклонение или плоскостность графика распределения,
size – количество генерируемых чисел.

Круговые диаграммы

pie(), т. к. они похожи на разрезаемый пирог.

Первым параметром является последовательность внесенных значений. После следуют необязательные аргументы:

explode – часто кусок ‘пирога’ выдвигают из центра. Эта последовательность имеет тот же размер, что и первый аргумент.

colors – задает цвета. По умолчанию для matplotlib это blue, green, red, cyan, magenta, yellow.

labels – это имена, у нас названия языков программирования.

labeldistance – определяет радиальные расстояния, на котором эти имена выводятся

autopct – задаёт, как форматируются численные значения

pctdistance – каком расстоянии от центра располагаются числовые значения

shadow – тень: boolean

plt.axes(0.0, 0.0, 1.0, 1.0) – одинаковые размеры по осям.

```

import matplotlib.pyplot as plt
x = [6, 12, 20, 7, 5, 5]
languages = ['Matlab', 'Java', 'Python', 'C', 'C++', 'Other']
plt.figure(figsize=(10,10))
explode = [0, 0, 0, 0.1, 0, 0]
plt.pie(x, labels = languages, explode=explode, autopct='%.1f%%', shadow=False)
plt.title('Circle diagram')
plt.show()

```

`figsize = (10,10)` – ширина и высота в дюймах

57. Списки. Сортировка. Сортировка вставками. Сортировка выбором.

Списки представляют собой **изменяемые последовательности**, обычно используемые для хранения коллекций однородных элементов, где степень сходства зависит от приложения.

В Python списки представлены **встроенным классом `list()`**, его можно использовать для преобразования итерируемых объектов в **тип `list`**.

Сортировка — это метод переупорядочения элементов или данных в порядке возрастания или убывания. В

Сортировка вставками (англ. *Insertion sort*) — алгоритм сортировки, в котором элементы входной последовательности просматриваются по одному, и каждый новый поступивший элемент размещается в подходящее место среди ранее упорядоченных элементов^[1]. Вычислительная сложность — $O(n^2)$.

```

import random as rd

a = [rd.randint(-1000, 1000) for _ in range(10)]
print('Изначальный список:', a, sep='\n')

# Сортировка вставками:
for i in range(1, len(a)):
    for j in range(i, 0, -1):
        if a[j] < a[j - 1]:
            a[j], a[j - 1] = a[j - 1], a[j]
        else:
            break
print('Отсортированный список:', a, sep='\n')

```

Сортировка выбором:

1. В неотсортированном подмассиве ищется локальный максимум (минимум).
2. Найденный максимум (минимум) меняется местами с последним (первым) элементом в подмассиве.
3. Если в массиве остались неотсортированные подмассивы — смотри пункт 1.

```

import random as rd

a = [rd.randint(-1000, 1000) for _ in range(10)]
print('Изначальный список:', a, sep='\n')

```

```

# Сортировка выбором:
for i in range(len(a) - 1):
    min_elem = a[i]
    min_ind = i
    for j in range(i + 1, len(a)):
        if a[j] < min_elem:
            min_elem = a[j]
            min_ind = j
    if min_ind != i:
        a[i], a[min_ind] = a[min_ind], a[i]

print('Отсортированный список:', a, sep='\n')

```

58. Списки. Сортировка вставками. Метод простых вставок. Метод вставок с бинарным поиском. Вставки с барьером. Метод Шелла.

Списки представляют собой **изменяемые последовательности**, обычно используемые для хранения коллекций однородных элементов, где степень сходства зависит от приложения.

В Python списки представлены [встроенным классом list\(\)](#), его можно использовать для преобразования итерируемых объектов в [тип list](#).

Сортировка — это метод переупорядочения элементов или данных в порядке возрастания или убывания. В

Сортировка вставками (англ. *Insertion sort*) — алгоритм сортировки, в котором элементы входной последовательности просматриваются по одному, и каждый новый поступивший элемент размещается в подходящее место среди ранее упорядоченных элементов^[1]. Вычислительная сложность — $O(n^2)$.

```

import random as rd

a = [rd.randint(-1000, 1000) for _ in range(10)]
print('Изначальный список:', a, sep='\n')

# Сортировка вставками:
for i in range(1, len(a)):
    for j in range(i, 0, -1):
        if a[j] < a[j - 1]:
            a[j], a[j - 1] = a[j - 1], a[j]
        else:
            break
print('Отсортированный список:', a, sep='\n')

```

Метод вставок с бинарным поиском:

```

import random as rd

a = [rd.randint(-1000, 1000) for _ in range(10)]
print('Изначальный список:', a, sep='\n')

# Сортировка вставками с бинарным поиском:
for i in range(1, len(a)):
    elem = a[i]
    left_border = 0
    right_border = i
    if left_border == right_border:
        left_border += 1
    else:

```

```

        while left_border < right_border:
            mid = (left_border + right_border) // 2
            if elem < a[mid] :
                right_border = mid
            else: left_border = mid + 1
        j = i
        while (j > left_border and j > 0):
            a[j] = a[j-1]
            j -= 1
        a[left_border] = elem

print('Отсортированный список:', a, sep='\n')

```

Сортировка вставками с барьером:

```

import random as rd

a = [rd.randint(-1000, 1000) for _ in range(10)]
print('Изначальный список:', a, sep='\n')

# Сортировка вставками с барьером:
def insertion_barrier_sorting(arr):
    arr = [0] + arr
    for i in range (1, len(arr)):
        arr[0] = arr[i]
        j = i - 1
        while arr[0] < arr[j]:
            arr[j+1] = arr[j]
            j -= 1
        arr[j+1] = arr[0]
    return arr[1:]
a = insertion_barrier_sorting(a)
print('Отсортированный список:', a, sep='\n')

```

Сортировка Шелла:

```

import random as rd

a = [rd.randint(-1000, 1000) for _ in range(10)]
print('Изначальный список:', a, sep='\n')

# Сортировка Шелла:
def shell_sort(data: list):
    last_index = len(data) - 1
    step = len(data) // 2
    while step > 0:
        for i in range(step, last_index + 1, 1):
            j = i
            delta = j - step
            while delta >= 0 and data[delta] > data[j]:
                data[delta], data[j] = data[j], data[delta]
                j = delta
                delta = j - step
        step //= 2
    return data
a = shell_sort(a)
print('Отсортированный список:', a, sep='\n')

```

59. Списки. Сортировка. Обменные методы сортировки. Сортировка пузырьком. Сортировка пузырьком с флагом. Метод шейкер-сортировки.

Списки представляют собой **изменяемые последовательности**, обычно используемые для хранения коллекций однородных элементов, где степень сходства зависит от приложения.

В Python списки представлены **встроенным классом list()**, его можно использовать для преобразования итерируемых объектов в тип **list**.

Сортировка — это метод переупорядочения элементов или данных в порядке возрастания или убывания. В

Метод «пузырька» или обменная сортировка — один из самых простых алгоритмов сортировки. Его название происходит от того, как работает алгоритм: с каждым новым проходом самый большой элемент «всплывает» к своему месту в упорядоченном **списке**.

Сортировка пузырьком:

```
import random as rd

a = [rd.randint(-1000, 1000) for _ in range(10)]
print('Изначальный список:', a, sep='\n')

# Сортировка пузырьком:
for run in range(len(a) - 1):
    for i in range(len(a) - 1 - run):
        if a[i] > a[i + 1]:
            a[i], a[i + 1] = a[i + 1], a[i]
print('Отсортированный список:', a, sep='\n')
```

Сортировка пузырьком с флагом:

```
import random as rd

a = [rd.randint(-1000, 1000) for _ in range(10)]
print('Изначальный список:', a, sep='\n')

# Сортировка пузырьком с флагом:
for run in range(len(a) - 1):
    flag = True
    for i in range(len(a) - 1 - run):
        if a[i] > a[i + 1]:
            a[i], a[i + 1] = a[i + 1], a[i]
            flag = False
    if flag:
        break
print('Отсортированный список:', a, sep='\n')
```

Шейкерная сортировка:

```
import random as rd

a = [rd.randint(-1000, 1000) for _ in range(10)]
print('Изначальный список:', a, sep='\n')
```

```

# Шейкерная сортировка:
left = 0
right = len(a) - 1

while left <= right:
    for i in range(left, right, +1):
        if a[i] > a[i + 1]:
            a[i], a[i + 1] = a[i + 1], a[i]
    right -= 1

    for i in range(right, left, -1):
        if a[i - 1] > a[i]:
            a[i], a[i - 1] = a[i - 1], a[i]
    left += 1

print('Отсортированный список:', a, sep='\n')

```

60. Списки. Сортировка. Метод быстрой сортировки.

Списки представляют собой ***изменяемые последовательности***, обычно используемые для хранения коллекций однородных элементов, где степень сходства зависит от приложения.

В Python списки представлены [встроенным классом list\(\)](#), его можно использовать для преобразования итерируемых объектов в [тип list](#).

Сортировка — это метод переупорядочения элементов или данных в порядке возрастания или убывания. В

Метод быстрой сортировки:

```

import random as rd

a = [rd.randint(-1000, 1000) for _ in range(10)]
print('Изначальный список:', a, sep='\n')

# Быстрая сортировка:
def quick_sort(s):
    if len(s) <= 1:
        return s
    rand_ind = rd.randint(0, len(s) - 1)
    elem = s[rand_ind]
    left = list(filter(lambda x: x < elem, s))
    center = [i for i in s if i == elem]
    right = list(filter(lambda x: x > elem, s))

    return quick_sort(left) + center + quick_sort(right)
a = quick_sort(a)
print('Отсортированный список:', a, sep='\n')

```

