Программирование. Подготовка к экзамену

От автора

Куда скидывать найденные очепятки и печенюшки Вы, думаю, уже знаете:

- t.me/zhikhkirill
- vk.com//zhikh.localhost
- github.com/zhikh23

Теоретические вопросы

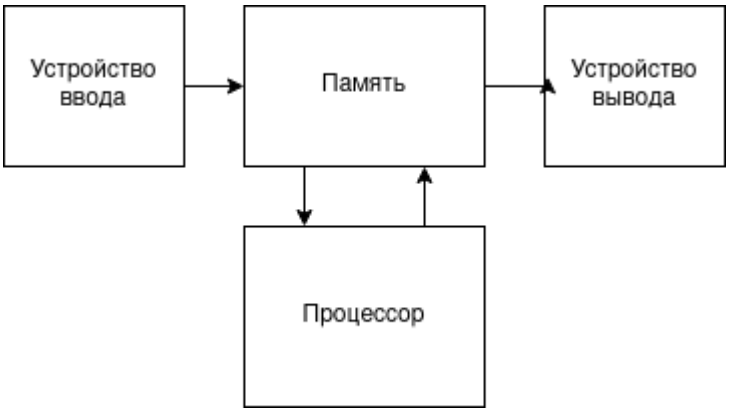
1. Электронная вычислительная машина. Устройство ЭВМ. Программа. Исходный текст, исполняемый файл

ЭВМ

ЭВМ -- основной вид реализации компьютеров, который технически выполнен на электронных элементах.

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

Устройство ЭВМ



> Примечание автора: сколько людей, столько и схем ЭВМ. На лекциях нам давали что-то похожее.

Программа и исходный текст

Исполняемая программа — сочетание компьютерных инструкций и данных, позволяющее аппаратному обеспечению вычислительной системы выполнять вычисления или функции управления.

Исходный текст программы — синтаксическая единица, которая соответствует правилам определённого языка программирования и состоит из инструкций и описания данных, необходимых для решения определённой задачи.

---

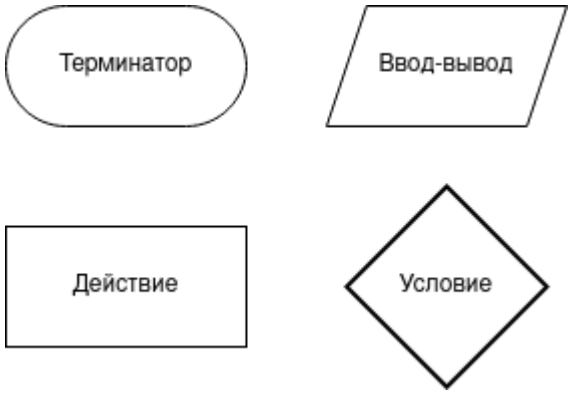
2. Схемы алгоритмов

Нестрогое определение

> От автора: PDF полной версии ГОСТа тут

Схема алгоритмов (она же *блок-схема*) -- схема, описывающая алгоритм или процесс в виде блоков различной формы, соединённых между собой линиями и стрелками.

Основные блоки



---

3. Языки программирования. Классификация

Язык программирования. Определение

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

- 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.

> Примечание автора: здесь и далее речь идёт о *третьей* версии -- `Python 3` (читается как *пайтон*). `Python 2` заметно отличается от последнего.

Структура программы

Программа -> Модули -> Операторы -> Выражения -> Объекты

Лексемы языка Python

Символы алфавита любого языка программирования образуют *лексемы*.

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

Всего существует пять видов лексем:

- ключевые слова (keywords)
  - Пример: `if`, `for`, `def` и т.п.
- идентификаторы (identifiers)
  - Пример: названия переменных, функций и т.п.
- литералы (literals)
  - Пример: `"hello world!"`, `42` и т.п.
- операции (operators)
  - Пример: `+`, `=`, `and`, `in` и т.п.
- знаки пунктуации (разделители, punctuators)
  - `,`, `;` и т.п.

---

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

Типы данных

Данные -- поддающееся многократной интерпретации представление информации в формализованном виде, пригодном для передачи, связи, или обработки.

Тип данных -- множество значений и операций над этими значениями.

Классификация

Основные способы классификации типов данных:

- скалярные и нескалярные;
- самостоятельные и зависимые (в том числе ссылочные).

Типы данных

Скалярные:

- Число -- `int`, `float`
- Логический тип -- `bool`

Нескалярные:

- Строка -- `str`
- Список -- `list`
- Словарь -- `dict`
- Кортеж -- `tuple`
- Множество -- `set`
- Файл

- Прочие основные типы
- Типы программных единиц
- Типы, связанные с реализацией

Приведение типов

Приведение типа -- преобразование значение одного типа в другое.

Бывает *явное* и *неявное*.

Неявное:

- 123 + 3.14

> Комментарий: здесь первое значение сначало *неявно* приводится к типу `float`, и лишь потом происходит сложение.

Явное:

- `int(3.14)`
- `str(obj)`

---

6. Операции над скалярными типами данных. Приоритеты операций

Над числами

- `x + y` -- сложение
  - `40 + 2 = 42`
- `x - y` -- вычитание
  - `16 - 2 = 14`
- `x * y` -- умножение
  - `16 * 2 = 32`
- `x / y` -- деление
  - `3 / 2 = 1.5`
- `x // y` -- целочисленное деление
  - `5 // 2 = 2`
- `x % y` -- остаток от деления
  - `5 % 2 = 1`
- `x ** y` -- возведение в степень
  - `2 ** 5 = 32`
- `-x` -- унарный минус
  - `x = 2; -x = -2`
- `+x` -- если бы мы знали, что это такое, но мы не знаем, что это такое
- `x | y` -- побитовое ИЛИ
  - `0b0101 | 0b0011 = 0b0111`
- `x & y` -- побитовое И
  - `0b0101 & 0b0011 = 0b0001`
- `x ^ y` -- побитовый ИСКЛЮЧАЮЩЕЕ ИЛИ
  - `0b0101 ^ 0b0011 = 0b0010`

- `~x` -- побитовое отрицание
  - `~0b0101 = 0b1010`
- `x << y` -- побитовый сдвиг влево
  - `0b11010110 << 2 = 0b01011000`
- `x >> y` -- побитовый сдвиг вправо
  - `0b11010110 >> 2 = 0b00110101`

Над логическими значениями

> Примечение автора: смотрим документацию и видим:
> *The `bool` class is a subclass of `int`*
> А значит для него определены *почти* (есть нюансы) все операции, что и для чисел. `True` эквивалетно `1`, а `False` -- `0`.

- `and` -- логическое И
- `or` -- логическое ИЛИ
- `not` -- логическое отрицание

Приоритеты операций

- `**`
- `~x`
- `+x`, `-x`
- `*`, `/`, `//`, `%`
- `+`, `-`
- `<<`, `>>`
- `&`
- `^`
- `|`
- `in`, `not in`, `is`, `is not`, `<`, `<=`, `>`, `>=`, `!=`, `==`
- `not x`
- `and`
- `or`

---

7. Функции ввода и вывода. Ввод данных

Ввод

```
# String
name = input("Enter your name: ")

# Integer
num = int(input("Enter integer number: "))

# Float
some_float_value = float(input())
```

```
# List
list_of_strings = input().split()
```

`prompt` -- текст-приглашение к вводу

Вывод

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

- `*objects` -- любое количество объектов, являющихся строками или поддерживающие приведение к ним (метод `__str__`).
- `sep` -- он же сепаратор. Строка, которая будет при выводе вставлена между отдельно переданными строками (см. пример ниже). По умолчанию -- пробел.
- `end` -- строка, которая будет добавлена в конец вывода. По умолчанию -- `\n`, т.е. перевод на новую строку.
- `file` -- куда будет напечатан результат. Обычно -- `sys.stdout`, `sys.stderr` или обыкновенный файл. По умолчанию -- `sys.stdout`.

```
# Examples
print("Hello, world!")

print("Hello", "world", sep=", ", end="!\n")    # Hello, world!

print("Error: something is wrong", file=sys.stderr)
```

---

8. Функции ввода и вывода. Функция вывода. Форматирование вывода

> Тоже самое, как и в вопросе 7

Форматирование вывода

```
replacement_field ::= "{" [field_name] ["!" conversion] [":" format_spec]
"}"
conversion ::= "r" | "s" | "a"
format_spec       ::= [[fill]align][sign][#][0][width][grouping_option]
["."precision][type]
fill              ::= any
align             ::= "<" | ">" | "=" | "^"
sign              ::= "+" | "-" | " "
width             ::= digit+
grouping_option   ::= "_" | ","
precision         ::= digit+
type              ::= "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" |
"n" | "o" | "s" | "x" | "X" | "%"
```

```
```

> От автора: "что ЭТО такое?!" -- это всего лишь простая форма Бэкуса — Наура (БНФ). Не волнуйтесь, её читать несложно. Особенно после регулярных выражений. Чтобы было ещё понятнее, покажу на примере.

Пример для `float`:

```
number = 42.0
print(f"{number:'='^12.5f}")  # out: '==42.00000=='
#        ^^^^^^  ^ ^ ^ ^^
# field_name _|  | | | ||
#          fill _| | | ||
#           align _| | ||
#             width _| ||
#            pecision _||
#                 type _|
```

- `field_name` -- что форматируем
- `fill` -- чем заполняем пустоты, которые образуются при выравнивании (`align`)
- `align` -- выравнивание `<`, `>`, `=`, `^`
- `width` -- ширина выравнивания
- `precision` -- точность указываемого значения для `float`
- `type` -- как форматировать

Пример для строк `str`:

```
s = "Hello!"
print(f"{s!r:-<12}")    # 'Hello!'----
#         ^
#         |_ conversion
```

- `conversion` -- как выводить строку (например, `r` экранирует все спец. символы и добавляет `'` на границах)

---

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. Обозначается `=`.

Множественное присваивание

Примеры:

```
```

```
a, b = "foo", "bar"

a, b = b, a

pos = (0, 4)
x, y = pos
```

Комбинированное присваивание

- `+=`
- `-=`
- `*=`
- `/=`
- `//=`
- `%=`
- `**=`
- `&=`
- `|=`
- `>>=`
- `<<=`

Выполняет действие над значеним и присваивает результут тому же имени.

---

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

Полный условный оператор

```
if expr1:
    do_1()
elif expr2:
    do_2()
else:
    do_else()
```

Неполный условный оператор. Пример

```
max_value = 0
if x > max_value:
    max_value = x
```

Тернарный оператор

```
result = value_1 if condition else value_2
```

Эквивалетно

```
if condition:
    result = value_1
else:
    result = value2
```

Пример:

```
max_value = x if x > y else y
```

---

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

Множественный выбор. Пример

Пример с некоторой реализацией меню:

```
cmd = input()
if not cmd:
    pass
elif cmd == "q":
    quit()
elif cmd == "m":
    menu()
elif cmd == "a":
    action()
else:
    print("Неизвестная команда")
```

Вложенные операторы условия. Пример

Пример с обработкой аргументов командной строки (почему бы и нет?)

```
arg = sys.argv[1]
if arg.startswith("--"):
    if arg == "--help":
        help()
    elif arg == "--interactive":
        run_interactive()
```

```
    elif arg == "--debug":
        debug()
    else:
        print(f"Неизвестный параметр:", arg)
        usage()
        exit(2)
elif arg.startswith("-"):
    if arg == "-h":
        help()
    elif arg == "-i":
        run_interactive()
    elif arg == "-d":
        debug()
    else:
        print(f"Неизвестный параметр:", arg)
        usage()
        exit(2)
```

---

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

Цикл. Определение

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

Операторы цикла

- `while`
- `for`

Цикл с условием

```
while condition_is_true:
    do_something()
else:
    do_if_no_breaked()
```

Операторы break и continue

`break` -- переходит за пределы ближайшего заключающего цикла (после всего оператора цикла)

```
while y < size:
    while x < size:   # <---+
        if x == 5:    #     | continue
```

```
            continue  # >---+
        print(x, y)
```

`conrinue` -- переходит в начало ближайшего заключающего цикла (в строку заголовка цикла)

```
while y < size:  # <--------+
    while x < size:   #     |
        if x == 5:    #     | break
            break     # >---+
        print(x, y)
```

---

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

Цикл. Определение

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

Операторы цикла

- `while`
- `for`

Цикл с итератором

```
for iterator in iterable:
    do_something()
else:
    do_if_no_breaked()
```

Функция range()

```
range(start = 0, stop, step = 1)
```

Порождает серию целых чисел `start <= n < stop` с шагом `step`.

> А вот тут в лекциях, очевидно, ошибка. Рабочий контр-пример ниже. Поэтому приведу своё определение
> Функция `range(start, stop, step)` возвращает объект, создающий последовательность чисел, начинающуюся с `start`, изменяемая каждую итерацию на `step` и останавливающуюся, когда достигает значения `stop`.

```
for i in range(5, -1, -1):
  print(i)
# 5, 4, 3, 2, 1, 0
```

---

14. Изменяемые и неизменяемые типы данных

Неизменяемые:

- `int`
- `float`
- `str`
- `bytes`
- `tuple`

Изменяемые

- `list`
- `dict`
- `set`
- и др.

Неизменяемые типы данных, как ни странно, не изменяемы: (`id` -- возвращает уникальный идентификатор объекта)

```
>>> a = 5
>>> id(a)
139709610098536
>>> a += 1
>>> id(a)
139709610098568
```

При попытке изменить неизменяемое значение, имени присваивается другой участок памяти (см. пример выше).

Изменяемые же ведут себя предсказуемо:

```
>>> l = [1, 2]
>>> id(l)
139709375880640
>>> l.append(3)
>>> id(l)
139709375880640
```

Из этого следует поведение неизменяемых и изменяемых объектов при передачу в функцию:

```python
def inc(x: int):
  x += 1

a = 5
inc(a)
print(a)    # 5


def append_one(x: list):
  x.append(1)

l = [1, 2]
append_one(l)
print(l)    # [1, 2, 1]
```

---

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

Список

Коллекция -- объект, содержащий в себе набор значений одного или различных типов и позволяющий обращаться к этим значениям.

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

> От автора: в `Python`-е, как ни странно, *списки реализованы в виде динамических массивов указателей*, а не как односвязанные списки. Больше можно прочитать тут.

Основные функции

- `all()` -- возвращает True, если все элементы истинны или список пуст
- `enumerate(iterable, start=0)` -- возвращает итератор последовательности кортежей (индекс, значение)
- `len(s)` -- количество элементов списка
- `max(iterable)`
- `min(iterable)`
- `print()`
- `reversed(seq)` -- возвращает итератор. Не создаёт копию последовательности. `b = list(reversed(a))`.
- `sorted(iterable, key = None, reverse = False)`
- `sum(iterable)`

Основные методы

- `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

- `+` -- конкатенация списков. Аналогично `extend`, но только для списков
- `*` -- "умножение" списка: `[ 0 ] * 5 => [ 0, 0, 0, 0, 0 ]`
- `in` -- принадлежность значения списку
- `del` -- удаление самого списка или его элемента
- `==` -- сравнение списков на совпадение элементов с учётом порядка
- `>`, `>=`, `<`, `<=` -- сравнение списков с учётом лексикографического порядка элементов

---

16. Списки. Создание списков. Списковые включения

Списки

См. ответ на вопрос №15

Списковые включения

Он же *генератор списков*:

```
l = [ value for iterator in iterable if condition ]

# Example
l = [ i**2 for i in range(5) ]  # [0, 1, 4, 9, 16]
```

Создание списков

- `l = []` или `l = list()` -- пустой список
- `l = [0] * 5` -- список с начальными значениями
- `l = [ i for i in range(5) ]` -- при помощи генератора списков

---

17. Списки. Основные методы для работы с элементами списка. Добавление элемента, вставка, удаление, поиск

Списки

См. ответ на вопрос №15.

Основные методы для работы с элементами списка

См. ответ на вопрос #15.

Добавление, вставка, удаление и поиск элемента

Добавление:

```
>>> l = list()
>>> l.append(5)
>>> l
[5]
```

Добавление в конец имеет сложность `O(1)`.

Вставка:

```
>>> l = [0, 2]
>>> l.insert(1, 1)
>>> l
[0, 1, 2]
```

Вставка имеет максимальную сложность `O(n)`.

Доступ по индексу

```
>>> l = [0, 2]
>>> l[1]
2
```

Доступ по индексы имеет сложность `O(1)` (помним, что списки в `Python`-е -- это массивы).

Удаление элемента по значению:

```
>>> l = [1, 2, 3]
>>> l.remove(2)
>>> l
[1, 3]
```

Удаление элемента по индексу:

```
>>> l = [1, 2, 3]
>>> l.pop(1)
2
>>> l
[1, 3]
```

Удаление значения имеет максимальную сложность `O(n)`.

Поиск:

```
>>> l = [1, 2, 3]
>>> found = None
>>> for i, it in enumerate(l):
...     if it == 2:
...         found = i
...         break
...
>>> if found is not None:
...     found
2
```

Или:

```
>>> l = [1, 2, 3]
>>> l.index(2)
1               # OR ValueError, if not found
```

Линейный поиск значения имеет максимальную сложность `O(n)`.

---

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

Списки

См. ответ на вопрос №15.

Основные операции со списками

См. ответ на вопрос #15.

Поиск минимального или максимального элемента

Поиск минимального:

```
```

```
from math import inf
l = [ ... ]
min_ = inf
for i, it in enumerate(l):
    if it < min_:
        min_ = it
        index = i
if min_ != inf:
    print(f"Min: l[{index}] = {min_}")
```

Поиск максимального:

```
from math import inf
l = [ ... ]
max_ = -inf
for i, it in enumerate(l):
    if it < max_:
        max_ = it
        index = i
if max_ != inf:
    print(f"Max: l[{index}] = {max_}")
```

Если нужно найти только само значение минимума/максимума (без индекса):

```
>>> l = [-1, 0, 1]
>>> min(l)
-1
>>> max(l)
1
```

Нахождение количества элементов

```
>>> l = [1, 2, 2]
>>> l.count(2)
2
```

Сумма и произведение элементов

Сумма:

```
>>> l = [1, 2, 3]
>>> sum(l)
6
```

Произведение:

```
l = [...]
prod = 1
for it in l:
    prod *= it
print(prod)
```

Или через модуль `functools` (не очень законно, зато красиво)

```
>>> import functools
>>> l = [1, 2, 3]
>>> functools.reduce(lambda prev, next: prev * next, l)
6
```

---

19. Списки. Использование срезов при обработке списков

Списки

См. ответ на вопрос №15.

Использование срезов

Срез -- объект, представляющий набор индексов, а также метод (способ), используемый для представления некоторой части последовательности

```
l[start:stop:step]
```

Рассмотреть список после определённого значения (например, обработка параметров командной строки)

```
import sys
program_path = sys.argv[0]
for argument in sys.argv[1:]:
    ...
```

> От автора: можно придумать ещё множество примеров... но пока пусть будет так.

---

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

Кортежи

Кортеж - неизменяемая последовательность (как список, только неизменяем).

Создание:

```
a = tuple() # Empty tuple
b = 1,      # Tuple with 1 element
c = (1, )
d = 1, 2    # (1, 2)
```

Функции, методы, операторы

Функции -- все как у списков. Методы -- `index(x)` и `count(x)` Операторы: -- `in`, `not in`

---

21. Словари. Понятие ключей и значений. Создание словарей. Основные функции, методы, операторы для работы со словарями

Словари

Словари -- неупорядоченные коллекции произвольных объектов с доступом по ключу.

> Примечание автора: это буквально *хэш-таблица*, если вы понимаете, о чём я

Каждому *ключу* соответствует единственное *значение*. Ключи обязательно должны быть *хэшируемыми* и *сравнимыми*.

```
>>> { []: 5 }
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Создание словарей

```
a = dict()  # Empty dictionary
b = { 1: "a" }
c = dict.fromkeys([1, 2], None)     # { 1: None, 2: None }
d = { a: a**2 for a in range(3) }   # { 0: 0, 1: 1, 2: 4 }
```

Основные операторы, функции и методы

Операторы:

- `del` -- удалить пару ключ-значение
- `in` -- проверить, есть ли ключ в словаре
- `not in`

- `|` -- расширить словарь другим словарём
- `|=` -- расширить словарь другим словарём и присвоить результат первому имени

Функции: Те же, что и для списков, только применяются к ключам (см. ответ на вопрос №15).

Методы:

- `clear()` - очищает словарь (удаляет все элементы)
- `copy()` - создаёт "мелкую" копию
- `fromkeys(iterable[, value])` - создаёт словарь на основе ключей и значения по умолчанию. Это *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

---

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

Множества

Множество (set) -- это неупорядоченная последовательность элементов, каждый из которых в множестве представлен ровно один раз.

Элементы множества должны быть *хэшируемыми*.

Основные функции, методы и операторы

Функции:

- `len(s)`

Методы:

- `isdisjoint(other)` -- `True`, если нет пересечения

```
>>> a = { 1, 2 }
>>> b = { 3, 4 }
>>> c = { 2, }
>>> a.isdisjoint(b)
True
```

```
>>> a.isdisjoint(c)
False
```

- `issubset(other)` -- `True`, если является подмножеством, <=

```
>>> a = { 1, 2 }
>>> b = { 3, 4 }
>>> c = { 2, }
>>> a.issubset(b)
False
>>> a.issubset(c)
False
>>> c.issubset(a)
True
```

- `issuperset(other)` -- `True`, если является надмножеством, >=

```
>>> a = { 1, 2 }
>>> b = { 3, 4 }
>>> c = { 2, }
>>> a.issubset(b)
False
>>> a.issubset(c)
True
>>> c.issubset(a)
False
```

- `union(*others)` -- объедение множеств, не изменяет их

```
>>> a = { 1, 2 }
>>> b = { 3, 4 }
>>> a.union(b)
{1, 2, 3, 4}
```

- `intersection(*others)` -- пересечение множеств

```
>>> a = { 1, 2 }
>>> d = { 2, 3, }
>>> a.intersection(d)
{2}
```

- `difference(*others)` -- те элементы, что не вошли во второе множество

```

```

```
>>> a = { 1, 2 }
>>> d = { 2, 3, }
>>> a.difference(d)
{1}
```

- `symmetric_difference(other)` -- симметричная разность множеств

```
>>> a = { 1, 2 }
>>> d = { 2, 3, }
>>> a.difference(d)
{1, 3}
```

- `copy()` -- "мелкая" копия множества
- `update(*others)` -- расширяет множество
- `intersection_update(*others)` -- эквивалетно `s1 = s1.intersection(s2)`
- `difference_update(*others)` -- эквивалетно `s1 = s1.difference(s2)`
- `symmetric_difference_update(other)` -- эквивалетно `s1 = s1.symmetric_difference(s2)`
- `add(elem)` -- добавляет элемент в множество
- `remove(elem)` -- удаляет из множества, может порождать исключение `KeyError`
- `discard(elem)` -- удаляет без исключения
- `pop()` -- удаляет и возвращает элемент множества (какой -- загадка вселенной...)
- `clear()` -- очищает множество

---

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

Строка

Строка (str) - тип данных, значениями которого является произвольная последовательность символов. Реализуется как массив символов. *Неизменяемы.*

```
s = 'some "text"'
s = "some 'text'"
s = """a lot of
text"""
s = ""      # Empty string
s = str()   # Empty string
```

```
s = str(object)
s = str(bytes, encoding="utf-8", errors="strict")
```

Основные функции, методы, операторы

Функции -- все как у списков.

Методы:

- `capitalize()` -- переводит первую букву в верхний регистр
- `casefold()` -- один из способов перевода в нижний регистр
- `center(width[,fillchar])` -- центрирует строку, дополняя пробелами с двух сторон
- `count(sub[,start[,end]])` -- считает неперекрывающиеся вхождения подстроки в строку
- `encode(encoding="utf-8",erros="strict")` -- кодирование в заданную кодировку
- `endswith(suffix[,start[,end]])` -- проверка на окончание одним из суффиксов
- `expandtabs(tabsize=8)` -- замена табуляций на пробелы
- `find(sub[,start[,end]])` -- поиск подстроки в строке
- `format()`
- `index(sub[,start[,end]])` -- аналогично find, но порождает исключение ValueError, если вхождений нет
- `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
- `isprintable()` -- все символы “печатные” или строка 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
- `partition(sep)` -- разделяет строку на части по первому вхождению разделителя, возвращает кортеж из 3-х элементов
- `removeprefix(prefix)` -- удаляет префикс
- `removesuffix(suffix)` -- удаляет суффикс
- `replace(old, new[, count])` -- заменяет все вхождения подстроки
- `rfind(sub[,start[,end]])` -- ищет подстроку справа
- `rindex(sub[,start[,end]])` -- ищет подстроку справа с исключением
- `rjust(width[,fillchar])` -- дополняет пробелами слева до ширины
- `rpartition(sep)` -- делит по разделителю, поиск справа
- `rsplit(sep = None, maxsplit=-1)` -- возвращает список слов (частей), поиск справа

- `rstrip([chars])` -- удаляет завершающие символы
- `split(sep=None, maxsplit=-1)` -- возвращает список слов (частей) по разделителю
- `splitlines([keepends])` -- делит на части по переводам строк
- `startswith(prefix[,start[,end]])` -- если начинается с префикса
- `strip([chars])` -- удаляет символы и из начала, и с конца
- `swapcase()` -- меняет регистр
- `title()` -- переводит первые буквы слов в верхний регистр
- `translate(table)` -- преобразование символов по таблице
- `upper()` -- переводит в верхний регистр
- `zfill()` -- дополняет строку нулями слева

Операторы:

- `+`
- `*`
- `in`
- `not in`

Срезы

Аналогично как у списков

---

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

Матрицы

Матрица -- двумерный массив.

Создание

Создание матрицы n x m:

```
matrix = [ [0]*m for _ in range(n) ]
```

Операции с матрицами

Получение элемента n-ой строки и m-ого столбца:

```
matrix[n][m]
```

Транспонирование матрицы

```

```

```
m = ...
for y in range(len(m)):
    for x in range(y, len(m)):
        m[y][x], m[x][y] = m[x][y], m[y][x]
```

> От автора: можно ещё что-то придумать

---

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

Матрицы

Матрица -- двумерный массив.

Обработка диагоналей

Главная диагональ

```
for i in range(len(m)):
    print(f"{m[i][i]:>8}", end=" ")
```

Побочная диагональ (для неквадратной матрицы считаем, что это диагональ из левого ниженего угла)

```
for i in range(len(m)):
    print(f"{m[len(m)-1 - i][i]:>8}", end=" ")
```

Обработка треугольных матриц

Верхнетреугольная матрица (над главной диагональю)

```
for y in range(len(m)):
    for x in range(y+1, len(m)):
        print(f"{m[y][x]:>8}", end=" ")
    print()
```

Нижнетреугольная матрица (под главной диагональю)

```
for y in range(len(m)):
    for x in range(y):
        print(f"{m[y][x]:>8}", end=" ")
    print()
```

---

26. Отладка программы. Способы отладки

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

При отладке требуется:

- узнавать текущие значения переменных;
- выяснять, по какому пути выполнялась программа.

Способы отладки:

- использование отладочной печати (отладочного вывода);
- использование отладчика.

---

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

Подпрограмма

Подпрограмма - поименованная или иным образом идентифицированная отдельная функционально независимая часть компьютерной программы.

Подпрограммы делятся на *процедуры* и *функции*.

Функции

Оператор `def` создаёт новый объект и присваивает его имени.

```
def <name_of_function>(<arguments>):
    ...
```

Пример:

```
def greet(name):
    print(f"Hello {name}!")

greet("Bob")    # out: Hello, Bob!
```

Аргументы функции

Присваивание *новых* значений аргументам внутри функций не затрагивает вызывающий код.

Модификация аргумента внутри функции:

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

Виды параметров в Python:

- позиционные аргументы
- именованные аргументы

```
#   positional args
#     v  v
def f(a, b, c=5, d=6):
#             ^    ^
#          named args
    ...
```

---

28. Функции. Области видимости

Функции

См. ответ на вопрос №27.

Области видимости

Область видимости (scope) -- это та часть кода, где переменная доступна, открыта и видима.

Области видимости:

1. *Глобальная* -- если переменная объявлена за пределами всех `def`, то она является "глобальной".
2. *Локальная* -- переменная, объявленная внутри `def`, будет локальной в своей фунции.
3. *Нелокальная* -- переменная, объявленная внутри `def`, включающем другие `def` (см. Замыкания).
4. *Встроенная* (built-in).

Оператор `global` делает имя внутри функции *глобальным*. Оператор `nonlocal` делает имя внутри функции *нелокальным*.

(*Правило LEGB*) Поиск имени выполняется последовательно в:

1. local
2. enclosing (см. *Нелокальная*)
3. global
4. built-in

---

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

Функции

См. ответ на вопрос #27.

Завершение работы функции

При помощи оператора `return`:

```
def greet(name):
    if not name:
        return
    print(f"Hello {name}!")
    return  # Optional
```

Исключения тоже завершают работу функции:

```
def greet(name):
    if not name:
        raise ValueError("empty name is not allowed")
    print(f"Hello {name}!")
```

Прямые, косвенные рекурсивные функции

Рекурсия -- вызов подпрограммы из неё же самой:

- непосредственно -- *прямая* 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*.

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* -- рекурсивный вызов является последним оператором функции.

---

30. Функции высшего порядка. Замыкания

Функции высшего порядка

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

```
def sum(iterable):
    res = 0
    for it in iterable:
        res += it
    return res
```

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

```
def map(iterable, function):
    res = list()
    for it in iterable:
        res.append(function(it))
    return res
```

Замыкания

Замыкание (closure) -- функция первого порядка, 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
```

> От автора: слишком "стерильный" пример. Приведу реальные участки кода из лабороторной

```
def create_console_output_channel() -> OutputChannel:
    """ Возвращает функцию, печатающую в окно консоли """
    size = get_console_size()
    def console_output_channel(text: str) -> None:
        """ Печатает текст в консоль с учётом ширины окна """
        text = format_alignment(text, size.width)
        print(text, end="")
    return console_output_channel

...

out = create_terminal_output_channel()
out(very_long_text)
```

Функция `console_output_channel` является *замыканием*, а переменная `size` -- нелокальной.

---

31. lambda-функции

---

32. Аннотации

---

33. Функции map, filter, reduce, zip

---

34. Декораторы

---

35. Знак “_“. Варианты использования

---

36. Модули. Способы подключения

---

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

---

38. Модуль time

---

39. Модуль random. Работа со случайными числами

---

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

---

41. Объектно-ориентированное программирование. Основные понятия ООП

---

42. Исключения

---

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

---

44. Файлы. Режимы доступа к файлам

---

45. Файлы. Текстовые файлы. Основные методы для работы

---

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

---

47. Файлы. Текстовые файлы. Итерационное чтение содержимого файла

---

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

---

49. Файлы. Оператор with. Исключения при работе с файлами

---

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

---

51. Модуль struct

---

52. Модуль os. Основные функции

---

53. Генераторы. *

---

54. Модуль numpy. Обработка массивов с использованием данного модуля. Работа с числами и вычислениями

---

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

---

56. Модуль matplotlib. Построение гистограмм и круговых диаграмм

---

57. Списки. Сортировка. Сортировка вставками. Сортировка выбором

---

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

---

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

---

60. Списки. Сортировка. Метод быстрой сортировки