Блоки кода

- Блоки ограничивают участок кода, принадлежащий управляющей конструкции
- Начинаются с “:”, которым оканчивается конструкция
- Все строки блока имеют уровень отступа равным начальной строке блока
- Отступы делаются с помошью табуляции или пробелов
- Блоки могут содержать другие блоки (с более глубокими отступами)

```
1       Some_contruction:
2           y = 2
3           z = x + y
4       #end_of_block
```

Блоки кода

- Блоки это не области видимости переменных. Переменные видны и после выхода из блока
- pass – пустой блок

if - Условное выполнение участков кода

```
1       if  condition1 :
2           pass # excuted if condition1 is true
3       elif condition2 :
4           pass # excuted if condition1 is false and condition2 is
5       #...
6       else:
7           pass # executed if all conditions is false
```

```
1       x = 12
2       sign = 0
3       if x > 0:
4           print x, "positive"
5           sign = 1
6       elif x < 0:
7           print x, "negative"
8           sign = -1
9       else:
10          print x, "== 0"
11          sign = 0
```

inline if

```
1       res = x if x >= 0 else -x
2       # res = (x >= 0 ? x : -x)
```

while

```
1       while condition:
2           pass # executed while condition is true
3       else:
4           pass # if no error or break in body
5
6       x = 1
7       while x < 100:
8           print x, "less␣than␣100"
9           x *= 2
```

for - цикл по множеству

```
1       for x in iterable:
2           func(x) # for each element in iterable
3       else:
4           pass # if no error or break in body
5
6       sum = 0
7       for x in range(100):
8           sum += x
9       print x  # 99 * 100 / 2
10
11      for i in range(n): # xrange(n)
12          pass
```

for undercover

```
1       for x in container:
2           f(x)
3
4       # some times equal to
5
6       _tmp = 0
7       while _tmp < len(container):
8           x = container[_tmp]
9           f(x)
10          _tmp += 1
```

break & continue как всегда

- break выходит из цикла
- continue переходит к следующей итерации

Задача

- нужно декодировать телефонный номера для АОН.
- По запросу АОНа АТС посылает телефонный номер, используя следующие правила:
- - Если цифра повторяется менее 2 раз, то она должна быть отброшена
- - Каждая значащая цифра повторяется минимум 2 раза
- - Если в номере идут несколько цифр подряд, то для обозначения «такая же цифра как предыдущая» используется идущий 2 или более подряд раз знак #
- Входящая строка 4434###552222311333661 => 4452136

list – Список (Массив)

- Упорядоченное множество элементов, доступ по номеру
- var = [1, 2, 3]
- Индексация arr[x]
- Срезы arr[frm:to:step]
  [arr[frm], arr[frm + step], ...,]
- Отрицательный индекс - отсчет от конца. x[-1]
- Отсутвие индекса - frm -> 0, to -> -1, step -> 1
- arr[::-1] - инверсия элементов
- arr[:] - копия

list – Список (Массив)

$$x = [0^{0}_{-6}, 1^{1}_{-5}, 2^{2}_{-4}, 3^{3}_{-3}, 4^{4}_{-2}, 5^{5}_{-1}]$$

x[2] == 2 [0, 1, 2, 3, 4, 5]

x[-2] == 4 [0, 1, 2, 3, 4, 5]

x[2:] == [2, 3, 4, 5] [0, 1, 2, 3, 4, 5]

x[-2:] == [4, 5] [0, 1, 2, 3, 4, 5]

x[1:-1] == [1, 2, 3, 4] [0, 1, 2, 3, 4, 5]

x[1:-1:2] == [1, 3] [0, 1, 2, 3, 4, 5]

x[::-1] == [5, 4, 3, 2, 1, 0]

list – Операции нам элементам и срезам

```
1       x = [3, 4, 5, 6]
2       x[::2] = [2, 2]  # x == [2, 4, 2, 6]
3       x[::2] = 2 # error
4       del x[1] # x == [3, 5, 6]
5       x = [1, None, True, ["123", 2.4]]
6       [1, 2, 3] + ["a", "b"] # [1, 2, 3, "a", "b"]
```

Методы списка

```
1   # arr.append(val)
2   [1, 2].append(3) == [1, 2, 3]
3
4   # arr.extend(arr2)
5   [1, 2].extend([2, 3]) == [1, 2, 2, 3]
6
7   # arr.pop()
8   x = [1, 2]
9   x.pop() == 2
10  print x # [1]pfdnhf
11
12  # arr.insert(pos, val)
13  [1, 2].insert(0, "abc") == ["abc", 1, 2]
14
15  [1, 2].index(2) == 1
16  [1, 2].reverse() == [2, 1]
17  [1, 2, 4, 1, 2, 4, 1, 1].count(1) == 4
18  x = [1, 3, 2]
19  x.remove(1) # x == [3, 2]
20  x.sort() # x == [2, 3]
```

Range

```
1       range(x) == (0, ..., x – 1)
2       range(x, y, z) == range(x)[:y:z]
```

assert

- assert expr[, msg]
- assert x == 1, "X␣should␣be␣equal␣to␣0"

Функции - минимум

```
1     def func_name1(param1, param2):
2         "documentation"
3         # block
4         x = param1 + param2
5         return x
6
7     def func_name2(param1, param2):
8         "documentation"
9         # block
10        x = param1 + param2
11        if x > 0:
12            return x
13        else:
14            return 0
```

Program template

```
#!/usr/bin/end python
# -*- coding:utf8 -*-
......

def main():
    res = 0
    .....
    return res

if __name__ == "__main__":
    exit(main())
```

Гномья сортировка

Гномья сортировка основана на технике, используемой обычным голландским садовым гномом (нидерл. tuinkabouter). Это метод, которым садовый гном сортирует линию цветочных горшков. По существу он смотрит на следующий и предыдущий садовые горшки: если они в правильном порядке, он шагает на один горшок вперёд, иначе он меняет их местами и шагает на один горшок назад. Граничные условия: если нет предыдущего горшка, он шагает вперёд; если нет следующего горшка, он закончил.

tuple – кортеж

- Константный список (но можно изменять элементы, если они не константные)

```
1       tpl = (1, 2)
2       tpl = 1,2
3       tpl[1] = 3 # error
4       tpl = (1, [2, 3, 4])
5       tpl[1].append(1) => (1, [2, 3, 4, 1])
6       (1) == 1
7       (1,) == (1,)
8       user, passwd = ("user", "qwerty")
```

dict - словарь

- Набор пар (ключ, значение), с быстрым поиском по ключу x = {1:2, "3":4}
- Только константные ключи (tuple - ok)
- Элементы неупорядоченны
- Нет срезов

```
1    x[1] == 2
2    x[2] #error
3    1 in x == True
4    x[17] = True
5    # x = {1:2, "3":4, 17:True}
```

dict – Словарь

```
1   x = {1:2, "3":"4"}
2   dict(a=1, b=2) == {"a":1, "b":2}
3   x.items() == [(1, 2), ("3", "4")]
4   x.values() == [2, "4"]
5   x.keys() == [1, "3"]
6   x.copy() == {1:2, "3":"4"}
7   x.setdefault(key, val) == val # if key not in x else x[key]
8   x.get(5, None) == None # if 5 not in x else x[5]
9   x.clear() # {}
10  x.update(y)
11  dict.fromkeys(keys, val) # {key[0]:val, key[1]:val, ..} defa
```

Алгоритм Шеннона — Фано

- Элементы выписывают в порядке убывания вероятностей.
- Делятся на две части, суммарные вероятности символов которых максимально близки друг другу.
- В префиксном коде для первой части алфавита присваивается двоичная цифра «0», второй части — «1».
- Полученные части рекурсивно делятся и их частям назначаются соответствующие двоичные цифры в префиксном коде.

ДЗ

- Написать строковые функции xfind, xreplace, xsplit, xjoin используя срезы строк (без применения других методов строк).
  xfind(s1, s2) == s1.find(s2)
  xreplace(s1, s2, s3) == s1.replace(s2, s3)
  xsplit(s1, s2) == s1.split(s2)
  xjoin(s, array) == s.join(array)
- Написать кодирование и декодирование файла по Хаффману. На диске есть файл с именем "input.txt". Его нужно прочитать, закодировать символы использую алгоритм Хаффмана и записать результат в output.bin. В решении должно быть две функции hf_encode(string) str->str, и hf_decode(string) str->str. Первая кодирует, вторая декодирует. Входными элементами для алгоритма являются отдельные байты файла.
- Написать интерпретатор подмножества языка forth.

  Программа на Forth состоит из набора команд(слов), некоторые из которых имеют параметры. Для хранения данных используется стек - команды получают свои операнды с вершины стека и туда же сохраняют результаты. В подмножестве 5 команд:

put значение - ложит значение на вершину стека. Значение может быть числом или строкой. Строка заключается в кавычки, внутри строки кавычек быть не может
pop - убирает значение с вершины стека
add - изымает из стека 2 значения, складывает их, кладет результат в стек
sub - изымает из стека 2 значения, вычитает их, кладет результат в стек
print - вынимает из стека 1 значение, печатает его.

```
put 3
put "asdaadasdas"
```

Каждая команда начинается с новой строки. Строки, начинающиеся с '#' - комментарии. Ваша программа должна содержать функцию eval_forth(), принимающую строку на языке forth и исполняющую ее. По умолчанию из main вызывать eval_forth("example.frt")
Пример, если в example.rft будет:

```
put 1
put 3
```

```
add
print
```

То программа должна напечатать '4'. Сложение имеет такой же смысл, как и в питоне. Вычитание для строк не определено, все входные данные проверять с помощью assert.