# 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
13
14      n = 121213
15
16      dividers = []
17      while n > 3:
18          for divider in range(2, int(n ** 0.5) + 1):
```

```
19              if n % divider == 0:
20                  break
21          else:
22              break
23          n //= divider
24          dividers.append(divider)
25
26      if n != 1:
27          dividers.append(n)
```

# 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

## Нет

- goto
- switch + case
- until
- dowhile, dountil

# with

```
with expression as var:
    block

# mostly the same as

var = expression
var.__enter__()

block

if error_happened:
    if var.__exit__(error_data):
        # pass_error_further
    else:
        # supress_error
else:
    var.__exit__()
```

использование with

```
1   with open(“rC:\xxx.”bin, "w") as fd:
2       fd.write“”(– * 100 + "\n")
3       fd.write“”(+ * 100 + "\n")
4
5   with open(“rC:\xxx.”bin, "r") as fd:
6       for line in fd:
7           print line
8
9   with db.cursor() as cur:
10      curr.execute(update_request_1)
11      curr.execute(update_request_2)
12      # commit or rollback
```

# List comprehension

```
1       res = [func(i) for i in some_iter if func2(i)]
2
3       res = ["{:.2f}".format(i ** 0.5)
4                   for i in [-1, 0, 1, 2, 3]
5                       if i >= 0]
6
7       res == ['0.00', '1.00', '1.41', '1.73']
8
9       res = [(i + 0j) ** 0.5 for i in [-1, 0 ,1, 2, 3]]
10      res = {func(i) for i in some_iter if func2(i)}
```

## 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
```

## Unit tests - find

```
1     assert find("abc", "b") == 1
2     assert find("abc", "b") == "abc".find("b")
3
4     assert find("abc", "a") == 0
5     assert find("abca", "a") == 0
6     assert find("dabca", "a") == 1
7     assert find("", "a") == -1
8     assert find("a", "a") == 0
9     assert find("ab", "abc") == 0
10    assert find("b" * 1000 + "abc", "abc") == 1000
11    assert find("b" * 1000 + "abc", "abcd") == -1
12
13    all_symbols = "".join([chr(i) for i in range(255)])
14    assert find(all_symbols, chr(100)) == 100
15
16    assert find("", "") == 0
17    assert find("", "") == "".find("")
```

## Program template

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

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

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

# 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.