

# DIRTY PYTHON 1.0

STONE 13TH PRESENTS

ЛЕКЦИЯ 2



DIRTY  
PYTHON

BEFORE AND AFTER CODING



c++



javascript



java



python



# ЧТО У НАС СЕГОДНЯ?

- Познакомимся с ветвлением
- Поговорим о приоритете логических операторов
- Покуражимся while силы есть
- Заглянем краем глаза в списки
- Подумаем про for
- Рассмотрим match case
- И на сладкое будет сахар
- А чтоб жизнь медом не казалась, познаем дзен)))

# АХ ЕСЛИ БЫ, АХ ЕСЛИ БЫ...

- Прошлый раз мы закончили на булевых типах данных, и теперь, когда мы знаем, что есть ложь, а что истина, мы можем переходить к ветвлению
  - На английском если будет if - на Python тоже
  - Базовая конструкция выглядит так:  
`if <условие>:`  
    тело ветвления
  - В принципе этого достаточно. Но есть два принципиальных момента:
  - 1) соблюдайте отступы, чтобы код выполнялся только при соблюдении условия прописанного в if он должен быть с отступом в 4 пробела. Все что находится на одном уровне с if выполняется независимо от него
  - 2) следует помнить, что в конечном итоге, условие сводиться к True или False, насколько бы сложной или простой конструкция не была



# IF

- Напоминаем, что 0, пустая строка, пустой список, кортеж, словарь, пустое множество и None – все это будет воспринято как False , а все остальное, как True
- Поэтому если у вас переменная var не входит в перечень выше, то код

```
if var!=0:
```

```
    print('что-то делаем')
```

аналогичен коду

```
if var:
```

```
    print('что-то делаем')
```

- Соответственно, если вам нужно, чтобы цикл выполнялся, когда у вас переменная равна нулю (или чему-то в понимании Python, означающего False), достаточно написать так:

```
if not var:
```

```
    print('что-то делаем')
```

```
1  number = 2
2
3
4  if number:
5      print('Попали в первый if')
6
7  if number % 2: # остаток целочисленного деления 2 на 2 равен нулю,
8      print('Попали во второй if')
9
10 if not number % 2: # а тут мы пишем если не ноль, то есть правда
11     print('Попали в третий if')
12
```

In: lection2 ×

↑ /usr/local/bin/python3.10 /Users/di/Desktop/DirtyPython/dirtyPython1/L

↓ Попали в первый if

Как видите во второй if

≡ Попали в третий if

мы не попали, а в

Process finished with exit code 0

третий – очень даже

# УСЛОВИЯ ДЛЯ IF

- Как уже упоминалось, условия для if могут быть довольно сложными. Но в конечном итоге, они сводятся к ответу True – и тогда код выполняется, или False – и тогда код не выполняется
- Примеры условий для if:
- if var
- if not var
- if var%2
- if var.isdigit()
- if var.isdigit() and 0 < int(number) < 11
- И даже еще более сложные конструкции

```
4 number = '7'  
5  
6 # print(bool(number))  
7  
8  
9 if number.isdigit():  
10     print('Зашли в тело IF')  
11  
  
D:\WorkShopPy\.my_venv_3.11.3\Scripts\python.exe D:\Work  
Зашли в тело IF
```

```
number = '7q'  
  
# print(bool(number))  
  
# print(0 == 2)  
  
if number.isdigit() and 0 < int(number) < 11:  
    print('Зашли в тело IF')
```



# ЧТО ИДЕТ ПОСЛЕ IF



```
4 list_contacts = [1, 2, 3]
5
6 # print(bool(number))
7
8 # print(0 < int(number) < 11)
9
10 if list_contacts:
11     print(list_contacts)
12
13 print('Что-то дальше после IF')
14
```

Run new ×  
D:\WorkShopPy\.my\_venv\_3.11.3\Scripts\python.exe  
[1, 2, 3]  
Что-то дальше после IF

```
4 list_contacts = []
5
6 # print(bool(number))
7
8 # print(0 < int(number) < 11)
9
10 if list_contacts:
11     print(list_contacts) ←
12
13 print('Что-то дальше после IF') ←
14
```

Run new ×  
D:\WorkShopPy\.my\_venv\_3.11.3\Scripts\python.exe  
Что-то дальше после IF  
Process finished with exit code 0

Эта строчка входит в тело цикла

А эта уже нет, и не зависит от выполнения условия

- Сравните код слева и справа:
- Слева условие выполнилось, потому что список не пуст, и он был напечатан. А затем выполнился код в 13 строкке
- Справа список пуст, а значит соответствует False, и выполнился только код в 13 строкке
- Таким образом, как только вы пишите код без отступа – тело цикла заканчивается



# ТАК ИЛИ ИНАЧЕ

- Иногда нам не достаточно, чтобы определенный код выполнялся при соблюдении условий, а нам хочется, чтобы что-то происходило в противоположном случае.
- В этом нам поможет иначе – else
- Это не обязательная часть ветвления, if будет работать и без него. Просто так удобнее.
- Else всегда относится к тому if с которым находится на одном уровне, и к тому который к нему ближе.

```
number = 3

if number:
    print('Попали в первый if')

if number % 2:
    print('Попали во второй if')

if not number % 2: | 💡
    print('Попали в третий if')
else:
    print('Наше условие не выполнилось')

else связан с ближайшим if, а не с теми, что выше
if not number % 2

: lection2 ×
↑ /usr/local/bin/python3.10 /Users/di/Desktop/DirtyP
↓ Попали в первый if
↓ Попали во второй if
↓ Наше условие не выполнилось
```

# ELIF



**if**  
**else**

- А что если у нас не один вариант развития событий?
  - Вы можете или делать ветвление внутри ветвления
  - Или использовать конструкцию `elif`
  - Как видно, его получили скрестив `else` и `if`, ну и работает примерно так: сначала проверяется условие `if`, если оно не выполняется – проверяется условие `elif` (их может быть несколько, проверяется по очереди), и если ни одно из условий не выполнилось переходится к `else`
  - `elif` как и `else` являются не обязательными, `if` вполне может существовать без них

```
13 #     print('Наше услов
14
15 guy = 'мудак какой-то'
16
17 if guy == 'богатый':
18     print('Подходящий')
19 elif guy == 'красивый':
20     print('зато красивый')
21 elif guy == 'с машиной':
22     print('эх... ну, хоть с работы забирать будет')
23 else:
24     print('мудак какой-то: ни денег, ни машины, еще и страшный...!')
25
26
27 else
28
29 lection2 x
30 ↑ /usr/local/bin/python3.10 /Users/di/Desktop/DirtyPython/dirtyPython1/Lec
31 ↓ мудак какой-то: ни денег, ни машины, еще и страшный...
32
33 Process finished with exit code 0
```

# ВЛОЖЕННОЕ ВЕТВЛЕНИЕ



- Но ничего не мешает вложить вам одно ветвление в другое. Это может быть в определенных случаях предпочтительнее, и сделает код более читаемым и понятным
- Хотя ветвлений внутри ветвления может быть несколько уровней (на самом деле вас никто не ограничивает) не следует чрезмерно увлекаться

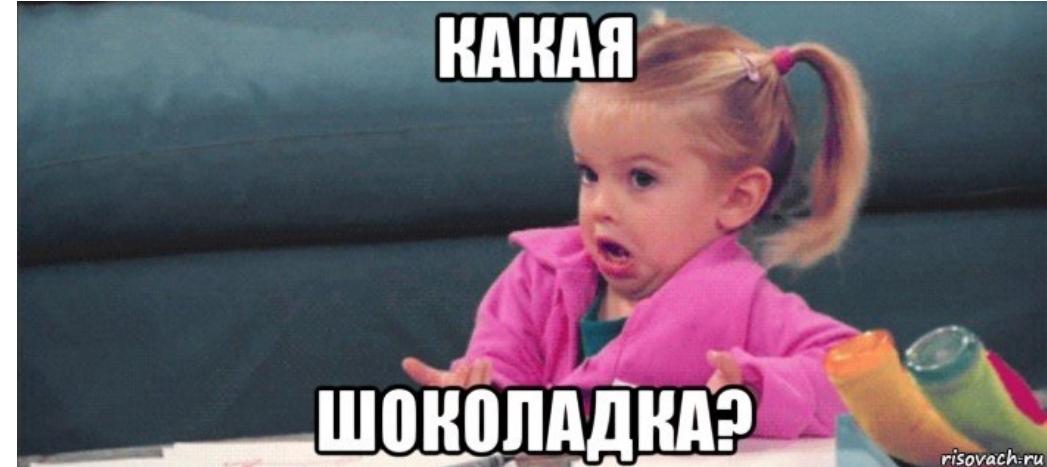
```
4     ball = 'RED'  
5  
6  
7     if ball == 'GREEN':  
8         print('Мяч зеленый')  
9     elif ball == 'RED':  
0         print('Мяч красный')  
1     elif ball == 'YELLOW':  
2         print('Мяч желтый')  
3     else:  
4         if ball == 'BLACK':  
5             print('Мяч черный')  
6         else:  
7             print('Мяч белый')  
8  
9     if ball == 'BLACK':
```

Вот эти if и else лежат внутри вон того else

# ЗАДАЧА С ШОКОЛАДКОЙ ИЛИ ПОЧЕМУ СКОБКИ ИМЕЮТ ЗНАЧЕНИЕ



- Сама задача: у вас есть количество долек по длине, количество долек по ширине, а также количество долек, которые нужно отломать от шоколадки. Нужно вывести ответ, можно ли отломать нужное количество долек от шоколадки за раз...
- Логическое решение такое: если общее количество долек, больше, чем требуется отломать, и количество отламываемых долек кратно длине или ширине шоколадки, то это возможно.
- И вот вам два варианта записи логического выражения



```
if length*width > count and count%length == 0 or count%width == 0:  
    print('Ура все получилось!')
```

```
if length*width > count and (count%length == 0 or count%width == 0):  
    print('Ура все получилось!')
```



# РАЗБОР

- Все дело в скобках! И приоритет логических операторов.
- Давайте запомним следующее
- and это логическое умножение
- or это логическое сложение
- А теперь давайте посмотрим результат различных действий

$1 * 1 = 1$  True

$1 + 1 = 2$  True

$1 * 0 = 0$  False

$1 + 0 = 1$  True

$0 * 1 = 0$  False

$0 + 1 = 1$  True

$0 * 0 = 0$  False

$0 + 0 = 0$  True

- Как и в математике, умножение имеет приоритет перед сложением. Именно поэтому нам нужны скобочки, чтобы все заработало правильно!



# ВСЕ ЕЩЕ ПРО СКОБКИ



```
18 length = 4
19 width = 3
20 count = 24

21
22 if length*width > count and count%length == 0 or count%width == 0:
23     print('Ура все получилось!')
24

D:\WorkShopPy\.my_venv_3.11.3\Scripts\python.exe D:\WorkShopPy\pythonProject\new.py
Ура все получилось!
```

```
13 length = 4
14 width = 3
15 count = 6

16
17 if length*width > count and (count%length == 0 or count%width == 0):
18     print('Ура все получилось!')

D:\WorkShopPy\.my_venv_3.11.3\Scripts\python.exe D:\WorkShopPy\pythonProject\new.py
Ура все получилось!
```

- Код слева отработал так, что у нас можно отломать 24 куска от шоколадки размером 4x3, потому как работал так:
  - 1)  $\text{длина} * \text{ширина} > \text{количество}$  (`False`) и количество кратно длине (`True`)  
-> `False and True -> False`
  - 2) Проверил кратность ширине -> `True`
  - 3) `False или True -> True`

- Код справа отработал бы корректно:
  - 1) Количество кратно длине (`False`) или количество кратно ширине (`True`) -> `True`
  - 2)  $\text{длина} * \text{ширина} > \text{количество}$  (`True`) и первое условие(`True`) -> `True`

# ЛЕННИВЫЙ IF



- Вроде в школе нас учили, что от перемены мест множителей произведение не меняется... Но логическое умножение немного отличается: вроде код одинаковый (только левая и правая части `and` поменялись местами), но слева у нас возникла ошибка, а справа нет. Почему?
  - Да потому, что если левая часть дает `False`, то правая часть даже не проверяется, ведь что ни умножь на ноль – получишь ноль. И таким образом Python экономит ресурс.

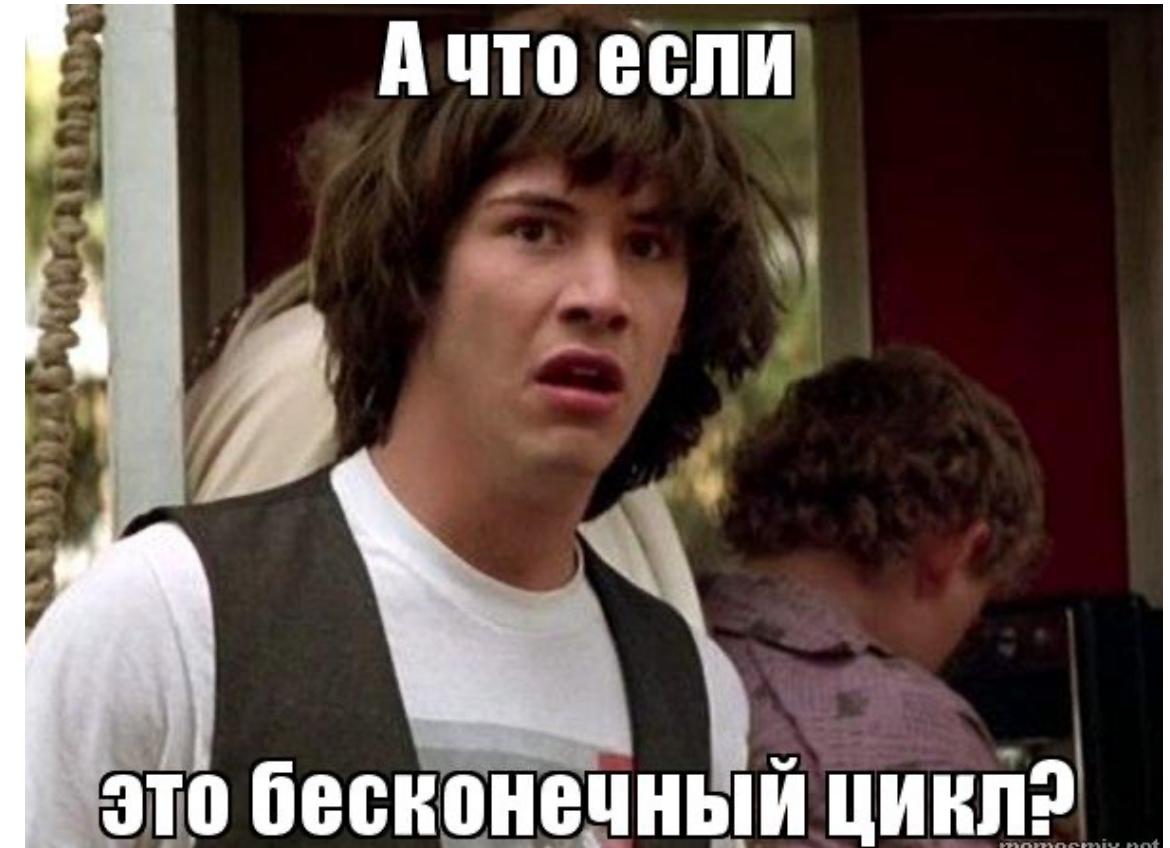
```
number = '7a'  
if 0 < int(number) < 10 and number.isdigit():  
    print('Ура все получилось!')  
  
Тут началось с проверки на то что  
число лежит в интервале от 0 до  
10, но поскольку number  
содержит букву, превратить его в  
число, обернув в int() не вышло
```

Тут началось с проверки на то что число лежит в интервале от 0 до 10, но поскольку number содержит букву, превратить его в число, обернув в int() не вышло

```
4 number = '7a'
5
6 |
7 if number.isdigit() and 0 < int(number) < 10:
8     print('Ура все получилось!')
9
10
11 А тут первой была проверка на
чило, получив False, правую часть
не выполнялась и ошибка не
возникла
```

# ЦИКЛЫ ЦИКЛЫ ЦИКЛЫ

- В Python есть следующие циклы:
- while - настоящий цикл, который работает, пока соблюдается условие
- for - скорее перебор, настроенные на определенное количество действий и работает с последовательностями, то есть с тем, что можно посчитать.
- Кстати, for всегда можно заменить на while, а наоборот не прокатывает. Поэтому мы начнем с while)
- И не бойтесь, с бесконечными циклами тоже разберемся.



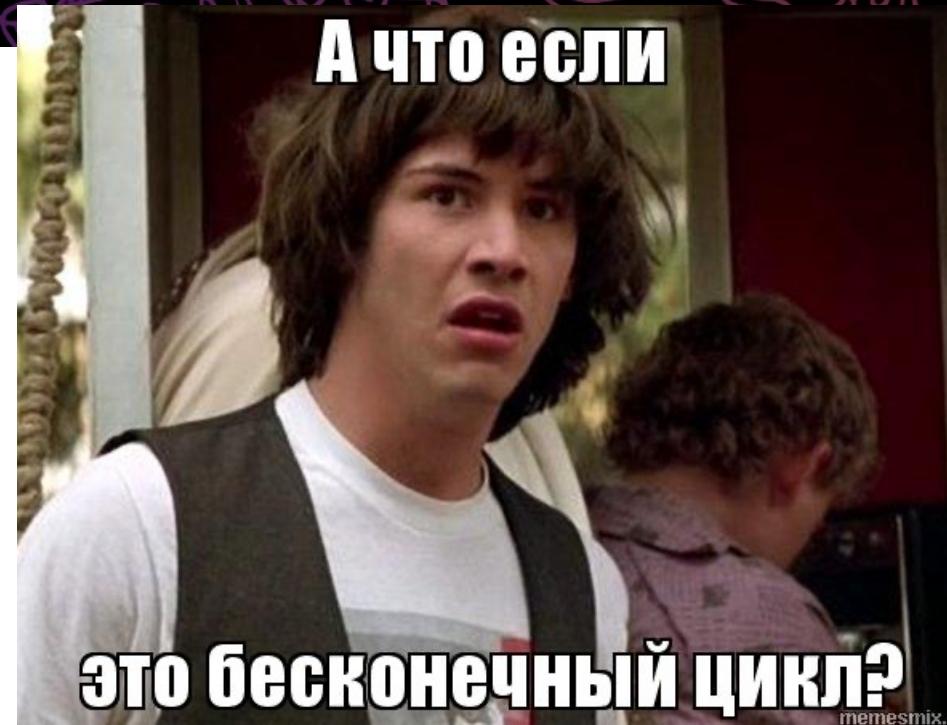
# WHILE

- Простейший while выглядит так:
- Назначаем i
- Ставим условие  $i < 10$
- Пишем тело цикла, не забыв про отступы
- Не забываем менять i ( $i += 1$  это тоже самое, что  $i = i + 1$ )
- Получаем 10 раз 'Ура'
- Но иногда нам выгодно получить бесконечный цикл, который будет выполняться, пока не выполниться определенное условие, которое мы не можем заранее предсказать (например, пользовательский ввод) бесконечный
- Вот тут бесконечный цикл, который будет хотеть от нас введения чего-то
- А тут бесконечный с условием прерывания

```

6  i = 0
7  while i < 10:
8      print('Ура')
9      i += 1
10

```



```

while True:
    number = input('Введите число: ')
    if int(number) == 10:
        break

```

```

while True:
    print('Ура')
    if i == 10:
        break
    i += 1

```

# КАК ВЫЙТИ ИЗ ЦИКЛА?



- Если вы были внимательны, то заметили, что было использовано выражение `break`. Оно дает вам возможность выйти из цикла при активации внешнего условия.

```
while True:  
    number = input('Введите число: ')  
    if number == '':  
        break
```

- Например в коде выше, цикл остановиться, как только будет введена пустая строка.
- Удобно? Очень удобно! Хотя тот же результат можно получить и не используя `while True`. Но для этого нам пришлось объявить переменную вне цикла, что не всегда выглядит красиво.

## Почему программа зависает?



```
number = 'wqer'  
while number != '':  
    number = input('Введите число: ')
```

# ЕЩЕ НЕМНОГО ПОВЫРАЖАЕМСЯ



- Выражение `continue` дает возможность пропустить часть цикла, где активируется внешнее условие, и вернуться к началу цикла.
- В этом коде мы будем вводить данные, если ввод не проходит проверку – срабатывает `print('это не число')`, а если проходит, то просто возвращается в начало. Кстати, перед `continue` тоже может быть код.
- Выражение `pass` позволяет обрабатывать условия без влияния на цикл. Фактически это заглушка, которая позволяет отложить написание куска кода на потом)

```
8     while True:
9         number = input('Введите число: ')
10        if number.isdigit():
11            continue
12        print('Это не число')
13
14
while True
Run new x
D:\WorkShopPy\my_venv_3.11.3\Scripts\python.exe D:\WorkShopPy\pythonProject\new.py
Введите число: 123
Введите число: 123
Введите число: 435
Введите число: 546765
Введите число: 345
Введите число: 6457
Введите число: вачрос
Это не число
Введите число:
```

# ЕЩЕ НЕМНОГО ПОВЫРАЖАЕМСЯ



- У циклов тоже может быть else! Часть кода, прописанная в else будет выполняться только, если цикл закончился без выполнения break, умер своей смертью, так сказать.

Как видите, тут выполнилось условие для break и цикл до конца не дожил...

```
6  i = 0
7  while i < 10:
8      if i == 7:
9          break
10     print('Ура!')
11     i += 1
12 else:
13     print('Цикл дожил до конца')

file i < 10
run new x
D:\WorkShopPy\.my_venv_3.11.3\Scripts\python.exe D:\Work
Ura!
Цикл дожил до конца

Process finished with exit code 0
```

```
6  i = 0
7  while i < 10:
8      if i == 11:
9          print('Наебнулось')
10         break
11     print('Ура!')
12     i += 1
13 else:
14     print('Цикл дожил до конца')

file i < 10
run new x
D:\WorkShopPy\.my_venv_3.11.3\Scripts\python.exe D:\Work
Ura!
Цикл дожил до конца

Process finished with exit code 0
```

А тут поменяли код так, что условие для break не выполнилось и цикл прошел все десять итераций, после чего выполнилось else

# FOR. НАЧАЛО

- Чтобы начать работать с циклами for, нам надо немного разобраться с тем, что же они могут перебирать.
- Перебирать можно то, что можно посчитать: символы в строке, элементы в списке... А вот данные типа int не годятся, потому как число оно и есть число.
- Но что делать, если нам нужно выполнить действие определенное количество раз? Например 10? Есть решение, надо завернуть в range(10). Чуть позже поговорим про range подробнее, но сначала познакомимся со списками





# СПИСКИ

- Списки – это тип данных, который представляет из себя коллекцию объектов, которые можно посчитать и поменять значения. В отличие от массивов – элементы списка могут быть самыми разными.
  - Несколько примеров:

my\_list1 =[1, 2, 3] список из int

```
my_list2 = [1, '2', 'abc', [3, 4], (0,)]    список разными элементами
```

- У списков довольно много методов, создавать их можно разными способами, но все это мы рассмотрим на следующей лекции – сегодня ровно столько, сколько нужно для работы с `for`

# СПИСКИ В ЦИКЛЕ FOR

- Итак, мы создали список из 10 нулей, а теперь в цикле for требуем 10 раз выполнить действие: присвоить  $i$ -тому элементу списка значение  $i$ . В результате у нас будет список от 0 до 9: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
- Но если мы напишем range(11), то получим ошибку, поскольку у нашего списка 11 элемента нет...
- И чтобы нам не пришлось расплачиваться за ошибки, можно воспользоваться методами списков:
- `my_list.append(< то что добавляем>)` добавляет в конец списка
- `my_list.insert(<куда добавляем>, < что добавляем>)` добавляет элемент по указанному индексу

```
my_list = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]  
  
for i in range(10):  
    my_list[i] = i
```

# СПИСКИ В ЦИКЛЕ FOR

- Посмотрим на примерах

```

27     my_list = []
28
29     for i in range(10):
30         my_list.append(i)
31     print(my_list)
32

```

Просто добавляем в конец списка

```

for i in range(10)

```

lection2 ×

```

↑ /usr/local/bin/python3.10 /Users/di/Desktop
↓ [0]
[0, 1]
[0, 1, 2]
[0, 1, 2, 3]
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4, 5]
[0, 1, 2, 3, 4, 5, 6]
[0, 1, 2, 3, 4, 5, 6, 7]
[0, 1, 2, 3, 4, 5, 6, 7, 8]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Process finished with exit code 0

```

```

6     my_list = []
7
8     for i in range(10):
9         my_list.insert(0, i)
10    print(my_list)
11
12

```

Добавляем по нулевому индексу

```

for i in range(10)

```

new ×

```

[0]
[1, 0]
[2, 1, 0]
[3, 2, 1, 0]
[4, 3, 2, 1, 0]
[5, 4, 3, 2, 1, 0]
[6, 5, 4, 3, 2, 1, 0]
[7, 6, 5, 4, 3, 2, 1, 0]
[8, 7, 6, 5, 4, 3, 2, 1, 0]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

```

```

6     my_list = []
7
8     for i in range(10):
9         my_list.insert(5, i)
10    print(my_list)
11
12

```

Добавляем по индексу 5:

До 5 идет вроде как обычно, но самое интересное появляется с пятым элементом

```

for i in range(10)

```

```

[0]
[0, 1]
[0, 1, 2]
[0, 1, 2, 3]
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4, 5]
[0, 1, 2, 3, 4, 6, 5]
[0, 1, 2, 3, 4, 7, 6, 5]
[0, 1, 2, 3, 4, 8, 7, 6, 5]
[0, 1, 2, 3, 4, 9, 8, 7, 6, 5]

```

# ПОГОВОРИМ ПРО RANGE



- Как мы уже говорили range позволяет сделать число итерируемым. Фактически range создает список от 0 до указанного числа (не включительно), который и перебирается в цикле for
- Но у range можно указать начало (по умолчанию 0), конец (не включительно) и даже шаг
  - for i in range(10) от нуля до 9
  - for i in range(2, 10) от 2 до 9
  - for i in range(0, 10, 2) каждый второй от 0 до 9
  - for i in range(-10, 10) от -10 до 9

```
33 for i in range(0, 10, 2):
34     print(i)
for i in range(0, 10, 2)
: lection2 ×
↑ /usr/local/bin/python3.10 /Users/di
↓ 0
= 2
= 4
= 6
= 8
```

The screenshot shows a code editor window with a Python script named 'lection2'. The script contains a for loop that prints even numbers from 0 to 10. The output of the script is shown in the terminal below the code editor, displaying the numbers 0, 2, 4, 6, and 8.

# ITEM

- Иногда возникает необходимость обращаться к каждомуциальному элементу в списке или каждомуциальному элементу в строке и что-то с ним делать. For для этого тоже прекрасно подходит. Теперь в цикле по очереди будут браться элементы (item) из списка. Item можно заменить на любое удобное слово – num, word, var...



```
my_list = ['1', 2, 'Три', 'IV']
for item in my_list:
    print(item)
```

D:\WorkShopPy\.my\_venv\_3.11.3\Scripts\python.exe D:\WorkShopPy\py

1  
2  
Три  
IV

# ВЫРАЖЕНИЯ ДЛЯ FOR

- Все те же конструкции, который мы изучили для while можно использовать и для for – break, continue, pass, else – могут существенно облегчить вам жизнь
- В этом коде выполнилось условие для break и поэтому цикл не дошел до конца и код в теле else не был выполнен.
- А вот этот дошел до else, т.к. условие для break не выполнилось
- И кстати, строку тоже можно использовать в цикле for, так как ее элементы прекрасно можно посчитать
- Следующий код посимвольно распечатает строку my\_string = 'abcdefg'

```
for symbol in my_string:
    print(symbol)
```

```

7  print(my_list)
8
9  for index in my_list:
10     print(index)
11     if index.isdigit():
12         print('Ебать! Цифра!')
13         break
14
15     else:
16         print('Умерло своей смертью')

for index in my_list : if index.isdigit():
Run new x
D:\WorkShopPy\my_venv_3.11.3\Scripts\python.exe D:\WorkShopPy\new.py
['1!', '2', 'Три', 'IV']
1!
2
Ебать! Цифра!
['1!', '2', 'Три', 'IV']

8  for index in 'asjdfandlqhdnbfas':
9      if index == 'w':
10         print('Чё, попалась, сучка!')
11         break
12     else:
13         print('К сожалению w нет в этом слове')
14

for index in 'asjdfandlqhdnbfas' : if index == 'w':
Run new x
D:\WorkShopPy\my_venv_3.11.3\Scripts\python.exe D:\WorkShopPy\pythonProject\new.py
К сожалению w нет в этом слове

```

# ФЛАГИ



- Флаги или триггеры нам вполне могут заменить break (например, если нам нужно знать выполнилось ли условие, а не прерывать цикл). У флагов булевые значения, в теле цикла значение флага меняется на противоположное при выполнении условия, а потом пишется еще один if, проверяющий состояние флага. Напомним, что if flag сработает, если flag у нас True

```
5
6
7 trigger = True
8 for index in 'asjdfandlqwhdnbfas':
9     if index == 'w':
10         print('Чё, попалась, сучка!')
11         trigger = False
12     if trigger:
13         print('К сожалению w нет в этом слове')
14
```



# ВСЕ ЕЩЕ ФЛАГИ..

- А тут у нас цикл чуть сложнее, но все с теми же флагами

```
36     trigger = True # ставим флаг в позицию True по умолчанию
37     count = 0        # до начала цикла заводим счетчик
38     for index in 'asdwesdswadswadsfw': # для символа в строке (ее сразу здесь можно прописать)
39         if index == 'w': # проверяем условие для смены флага
40             count += 1 # увеличиваем счетчик
41             print(f'Че, попалась? уже {count} раз') # печатаем результат
42             trigger = False # ставим флаг в позицию False так как условие выполнилось
43             if trigger: # если флаг остался True выполняем тело кода
44                 print('Нам такое не попадалось!')
45
if trigger
Run: lection2 ×
▶  ↗ /usr/local/bin/python3.10 /Users/di/Desktop/DirtyPython/dirtyPython1/Lection/lection2.py
↓  Че, попалась? уже 1 раз
↓  Че, попалась? уже 2 раз
↓  Че, попалась? уже 3 раз
↓  Че, попалась? уже 4 раз
```

Кстати, есть такая удобная штука как f строка. Она позволяет интегрировать значение переменной или выражения сразу в текст print Для этого перед кавычками нужно поставить f, а переменную или выражение обрамить фигурными скобками

# ЦИКЛЫ В ЦИКЛАХ



- Если помните, никто не мешает вам сделать ветвление внутри ветвления и ветвление внутри цикла. Так вот! Циклы в циклах тоже бывают. И уровень вложенности не ограничен. Единственное, что нельзя использовать одну и туже *i* (item) для разных циклов.
- Это может пригодиться при работе со вложенными списками, например. Или когда мы хотим получить таблицу умножения... Или все варианты возможных комбинаций символов

```
46 count = 0
47 for i in range(1, 4):
48     for j in 'abc':
49         for k in ['+', '-']:
50             print(i, k, j)
51             count += 1
52
53 print(f'{3*3*2} = {count}')
```

Run: lection2 ×

```
1 + c
1 - c
2 + a
2 - a
2 + b
2 - b
2 + c
2 - c
3 + a
3 - a
3 + b
3 - b
3 + c
3 - c
```

# MATCH CASE

- Разберем одну интересную конструкцию, которая вполне может заменить множество elif, да и выглядит красивее Match case (на самом деле она может много чего, но об этом отдельно)
- После match пишем переменную или выражение, которое будем проверять на точное совпадение
- В теле прописываем case – варианты, чему оно должно быть равно. Если будет совпадение – case выполниться
- В конце пишем case \_: для всех остальных непредусмотренных случаев



```
17 match number:  
18     case 1:  
19         print('Один')  
20     case 2:  
21         print('ДВА')  
22     case 3:  
23         print('ТРЫ')  
24     case 4:  
25         print('ЧИТИРЕ')  
26     case _:  
27         print('Я больше 4 не умею')
```



# ВОТ ВАМ МАТЧ CASE С ВЫРАЖЕНИЕМ

- Вот так например можно проверить на четность)))
- Давайте переведем выражение на русский язык: если число number равно нулю используй для проверки 'Zero', иначе используй результат целочисленного деления number на два (что даст нам 0 или 1)
- Далее идут case для 'Zero', 0, 1 и \_

```
for number in some_list:  
    match ('Zero' if not number else number % 2):  
        case 0:  
            print("even")  
        case 1:  
            print("odd")  
        case 'zero':  
            print("Zero")  
        case _:  
            print("get lost")
```

Кто заметил ошибку в коде?  
Правильно, Zero должно быть в case тоже с большой буквы

# ДЗЕН РУТНОН



- И что же выбрать бедному начинающему кодеру – матчкэйс или элиф? Что вообще значит – стиль написания? Как сделать красиво?
- Для этого надо познакомиться с Дзен Пайтона: филосовская концепция, которая отвечает на эти вопросы.
- Где почитать этот Дзен?
- Можно вот тут  
[https://ru.wikipedia.org/wiki/Дзен\\_Пайтона](https://ru.wikipedia.org/wiki/Дзен_Пайтона)
- А можно в Пайчарме выполнить `import this`

```
1 import this
2
3 run: lection2
4
5 /usr/local/bin/python3.10 /Users/di/Desktop/DirtyPython/dirtyPython1/Le
6
7 The Zen of Python, by Tim Peters
8
9
10 Beautiful is better than ugly.
11 Explicit is better than implicit.
12 Simple is better than complex.
13 Complex is better than complicated.
14 Flat is better than nested.
15 Sparse is better than dense.
16 Readability counts.
17 Special cases aren't special enough to break the rules.
18 Although practicality beats purity.
19 Errors should never pass silently.
20 Unless explicitly silenced.
21 In the face of ambiguity, refuse the temptation to guess.
22 There should be one-- and preferably only one --obvious way to do it.
23 Although that way may not be obvious at first unless you're Dutch.
24 Now is better than never.
25 Although never is often better than *right* now.
26 If the implementation is hard to explain, it's a bad idea.
27 If the implementation is easy to explain, it may be a good idea.
28 Namespaces are one honking great idea -- let's do more of those!
```



# СИНТАКСИЧЕСКИЙ САХАР

Что же такое синтаксический сахар?

Это решение одной задачи разными способами и конструкциями.

Весь код по заполнению списка цифрами можно написать в одну строку. Но не всегда нужно.

Но наша задача не выработать супер-почерк, а разобраться с основами.

Итак, сахар – конструкция, которая позволяет нам уменьшить код:

`count = count + 1 → count += 1`

Но сахар надо применять грамотно, а то жопа слипнется.

И мы с вами еще попробуем однострочники, но надо помнить про Дзен

# И ЧТО ДАЛЬШЕ?

- МЫ СЕГОДНЯ ХОРОШО ПОРАБОТАЛИ!
- НО БЕЗ ЗАКРЕПЛЕНИЯ НАВЫКОВ ЛЕКЦИИ МАЛО ЧТО ДАДУТ
- ПОЭТОМУ КОДИМ, КОДИМ И ЕЩЕ РАЗ КОДИМ
- РЕШАЕМ ЗАДАЧКИ И ТЕ ЧТО ДАЮТ НА DIRTY PYTHON, И ТЕ ЧТО ДАЮТ В ГБ, И ТЕ ЧТО НАЙДЕМ ГДЕ-ТО ЕЩЕ



DIRTY  
PYTHON