25. Конспект

Что такое цикл

Цикл — это управляющая конструкция для многократного исполнения инструкций.

В Python цикл состоит из оператора for или while, условия и тела цикла.

Условие — выражение, которое определяет, повторится цикл или нет.

Тело цикла — это блок кода с отступом, набор инструкций, который выполняется каждый раз.

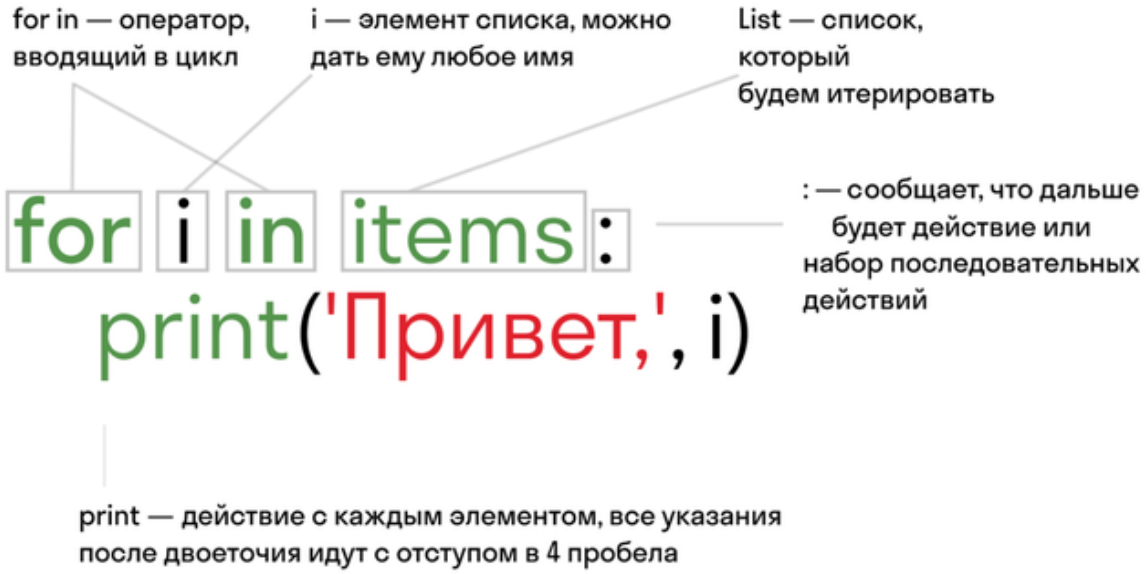
Связанные с циклом понятия — итерация и временная переменная.

Итерация — одно повторение цикла.

Временная переменная — переменная, которая меняет свое значение в каждой итерации.

Визуализировать пошаговое выполнение цикла можно на сайте https://pythontutor.com/

Цикл for



Задачи, которые можно решить с помощью цикла for

- Вывести все элементы списка:
- `letters = ["P","y","t","h","o", "n"]`
- `for item in letters:`

  `  print(item)`

  Выведет:

  ```
  P
  y
  t
  h
  o

  n
  ```

- Пронумеровать элементы списка:
- `reasons = ["ты", "все твои мечты", "боль"]`
- `num = 1`
-
- `for reason in reasons:`
- `  print(f"{num} причина — это {reason}")`

  `  num += 1`

- Просуммировать все элементы списка:
- `items_sum = 0`
- `numbers = [100, 200, 300]`
-
- `for number in numbers:`
- `  items_sum += number`
-

  `print(items_sum)`

- Вывести часть элементов списка:
- `items_list = ["f","f","g","h"]`
-
- `for item in items_list:`
- `    if item != "f":`

  `        print(item)`

  Выведет:

  ```
  g
  ```

```
h
```

Цикл while

Задачи, которые можно решить с помощью цикла while

- Посчитать от 0 до 4 помощью while:

```
i = 0
while i < 5:
  print("i =", i)

  i += 1
```

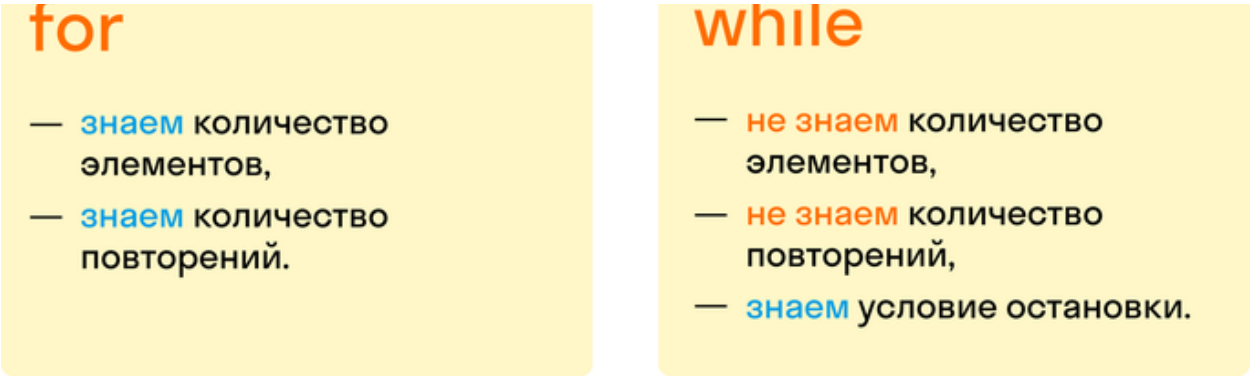
- Запустить бесконечный цикл:

```
answer = None

while answer != 'пошли':
  answer = input("Пошли тусить? ")

print("Я знал, что ты согласишься!")
```

В чем разница между циклами



Оператор break

```
while True:
    print("цикл повторяется бесконечно")
    print("чтобы остановить его, нужно убить программу")
    print("или использовать break")
```

Как это использовать:

```
# Абсолютно честно случайно выбранное число
num = 4

while True:
  ans = input("Угадай число ")
  if int(ans) == num:
    print("Угадал!")
    break
  else:
    print("Неа")
```

Пример, как создать бесконечный цикл и выйти из него:

```
repeat = True

while repeat == True:

    print("Скажите стоп, и мы закончим")

    command = input("")
    if comand == "стоп":
       break
    else:
       print("Продолжаем")
```

Конструкция for in range

Задачи, которые можно решить с помощью конструкции for in range

- Запустить цикл от x до y:
- ```
  for num in range(1, 4):

    print(f"смеяться {num} минуту")
  ```

  Выведет:

  ```
  смеяться 1 минуту
  смеяться 2 минуту

  смеяться 3 минуту
  ```

  Обратите внимание, что верхняя граница не включается. И по умолчанию, если стартовое число не указано, начинаем от нуля.

  Еще один пример:

  ```
  for num in range(5):

    print(num)
  ```

  Выведет:

  ```
  0
  1
  2
  3

  4
  ```
  ```

- Перебрать список, перебирая индексы:

```
actions = ['buy it', 'use it', 'break it', 'fix it']

action_indexes = range(len(actions)) # [0, 1, 2, 3]

for act_idx in action_indexes:

    print(actions[act_idx])
```

Выведет:

```
buy it
use it
break it

fix it
```

- Пронумеровать список, перебирая индексы:

```
girls = ["Иветта", "Лизетта", "Мюзетта"]

for i in range(len(girls)):

  print(i+1, girls[i])
```

Выведет:

```
1 Иветта
2 Лизетта

3 Мюзетта
```

Обратите внимание, что индексы в списках нумеруются с 0, а не с 1.

Поэтому когда нужна обычная нумерация, мы прибавляем к индексу 1.

Если добавить функции range() еще один аргумент, то **временная переменная будет увеличиваться не на 1, а на указанное число**.

Например:

```
for i in range(0, 8, 2) :
    print(i)
```

Выведет:

```
0
2
4
6
```

Если добавить функции range() третий аргумент и сделать его отрицательным, то **считать можно в обратном порядке**:

```
for i in range(10, 0, -2) :
    print(i)
```

Выведет:

```
10
8
6
4
2
```

Оператор continue

Continue используется как и break, но не прерывает цикл, а завершает текущую итерацию — сразу запускается следующая.

Оператор continue вызывает немедленный переход в начало цикла. Иногда он позволяет избежать вложения операторов.

Рассмотрим пример, где оператор continue используется для пропуска вывода нечетных чисел.

Код выводит все четные числа, которые меньше 10 и больше или равны 0:

```
x = 10
while x:
    x = x - 1  # Либо x -= 1
    if x % 2 != 0: continue # Нечетное? Тогда пропустить print
    print(x, end=' ')
```

0 означает ложь, а % — операцию получения остатка от деления, поэтому данный цикл делает обратный отсчет до 0, пропуская числа, которые не являются множителями 2.

Результат, который выведет:

```
8 6 4 2 0
```

List Comprehensions для работы со списками

List comprehensions — это еще один способ работать с итерируемыми объектами, например списками, который позволяет перейти от создания списков с помощью циклов к более компактным конструкциям.

Итерируемый (или перебираемый) объект — объект, который способен возвращать элементы по одному. Итерируемыми являются списки, словари, кортежи, множества и строки.

Булевы объекты и числа — неитерируемые объекты.

Создание списков с помощью list comprehensions

Рассмотрим работу list comprehensions на примере. Вместо конструкции типа:

```
a = []
for i in range(1,10):
    a.append(i)
```

Мы можем написать следующее:

```
[item for item in range(1,10)]
```

Результат будет одинаковым:

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

Конструкция

```
[i for i in range(1,15)]
```

— это list comprehension, или генератор списка. Квадратные скобки здесь указывают на то, что будет создан список.

В целом это такой синтаксический сахар, который позволяет быстрее и компактнее писать код.

Синтаксический сахар — возможности языка, применение которых не влияет на поведение программы, но делает использование языка более удобным для человека.

Перебор списков с list comprehensions

Общая схема для перебора списков выглядит так:

```
[<что будет в списке> for <временная переменная списка> in
<список>]
```

Проверим, как это работает. Допустим, у нас есть задача — прибавить 1 к каждому элементу списка.

Можно сделать это с помощью append(), и мы получим список, в котором каждый элемент возрастет на 1:

```
items = [1,2,3]
items_plus = []

for item in items:
      items_plus.append(item + 1)

print(items_plus)
>>> [2, 3, 4]
```

Либо, чтобы не тратить место, можно написать так:

```
items = [1,2,3]

items_plus = [item +1 for item in items]

print(items_plus)
>>> [2, 3, 4]
```

Следующий пример — возведение в степень.

Вместо такой записи:

```
items = [1,2,3]
items_exp = []

for item in items:
      items_exp.append(item ** 2)

print(items_exp)
>>> [1, 4, 9]
```

Напишем вот так:

```
items = [1,2,3]

items_exp = [item ** 2 for item in items]

print(items_exp)
>>> [1, 4, 9]
```

Преобразование типов

Частный случай преобразования списка — массовое изменение типа данных.

Например, у нас есть список строк, а мы хотим их привести к числам. Тогда вместо такой записи:

```
items = ["1", "2", "3"]

items_int = []

for item in items:
    items_int.append(int(item))

print(items_int)
>>> [1, 2, 3]
```

Сделаем так:

```
items = ["1", "2", "3"]

items_int = [int(item) for item in items]

print(items_int)
# Выведет [1, 2, 3]
```

Фильтрация списков

С помощью list comprehensions можно отфильтровать список и получить список поменьше.

Например, найдем все положительные числа из списка:

```
numbers = [-2, -1, 0, 1, 2]

numbers_positive = []

for num in numbers:
    if num > 0:
        numbers_positive.append(num)

print(numbers_positive)
>>> [1, 2]
```

Теперь с помощью list comprehensions:

```
numbers = [-2, -1, 0, 1, 2]

numbers_positive = [num for num in numbers if num >0 ]

print(numbers_positive)
```

```
>>> [1, 2]
```

Общая схема с условием выглядит так:
```
[<что будет в списке> for <временная переменная> in <список> if <условие>]
```

Вызов функций

Можно вызывать функции и внутри list comprehensions. Например, print.

Так можно распечатать список в одну строку:
```
items = [[1], [1, 2], [1, 2, 3]]

print([len(item) for item in items])
>>> [1, 2, 3]
```

Pass

При использовании операторов if/else/for/while код или часть кода пишется со сдвигом относительно самого оператора.

Такой код называют **блоком условия или цикла**. Для циклов еще распространено название **тело цикла**.

Пример:
```
i = 0

while i < 4:
    print(i) # Этот код в блоке/теле цикла
    i += 1   # Этот код в блоке/теле цикла
```

Особенность Python в том, что мы не можем написать условие или цикл, который не имеет блока, — получим ошибку отступов: **IndentationError**.

С такой ошибкой можно столкнуться, если выполнить подобный код:
```
i = 0

while i < 4:

print("Вот и все")
```

Если мы хотим написать условие или цикл, но еще не знаем, что будет в его блоке, для этого есть специальный оператор **pass** — плейсхолдер для будущего кода:

```
i = 0

while i < 4:
     pass

print("Вот и всё")
```

Или:

```
for n in [1,2,3,4,5]:
    pass
```

Или:

```
if user_is_active:
    pass
else:
    pass
```

В наших задачах на программирование можно будет часто встретить **pass** — это означает «тут будет код, нужно написать его, вперед».

Enumerate

Все циклы, с которыми мы работали до этого, меняли значение одной временной переменной. Нам это было удобно почти всегда, кроме задач, когда нужно одновременно работать с индексами и значениями.

Например, у нас есть список и мы хотим его пронумеровать:

```
letters = ["A", "B", "C", "D", "E", "F"]
```

Чтобы вывод был таким:

```
0 A
1 B
2 C
```

Мы обычно пишем такой код:

```
for i in range(len(letters)):
  letter = letters[i]
  print(i, letter)
```

Однако в Python есть множество удобных функций для типовых задач. В этом случае интерпретатор предлагает нам функцию **enumerate**.

Функция enumerate (от англ. «пронумеровать») позволяет перебирать не одну переменную, а сразу несколько, где в первую временную переменную попадает индекс (порядковый номер начиная с 0), а во вторую — значение.

Пример:
```
for i, letter in enumerate(letters):
  print(i, letter)
```

Выведет:
```
0 A
1 B
2 C
```

Теперь код стал более читаемым, хотя работает с теми же переменными, как и тот, который мы писали до этого.

Еще одна возможность enumerate — указывать стартовое значение **start** для индекса, так что самому первому элементу может соответствовать не 0, а 1, 2 или 10!
```
for i, letter in enumerate(letters, start=1):
  print(i, letter)
```

Выведет:
```
1 A
2 B
3 C
```

Разберем несколько примеров.

Пример 1. Выведем позиции элементов равных True:
```
items = [True, True, False, True, True, False]

for index, value in enumerate(items, start=1):
   if value:
       print(index)
```

Выведет:
```
1
2
```

```
4
5
```

Пример 2. Выведем только четные элементы:

```
letters = ["Alpha", "Bravo", "Charlie", "Delta", "Echo",
"Foxtrot"]

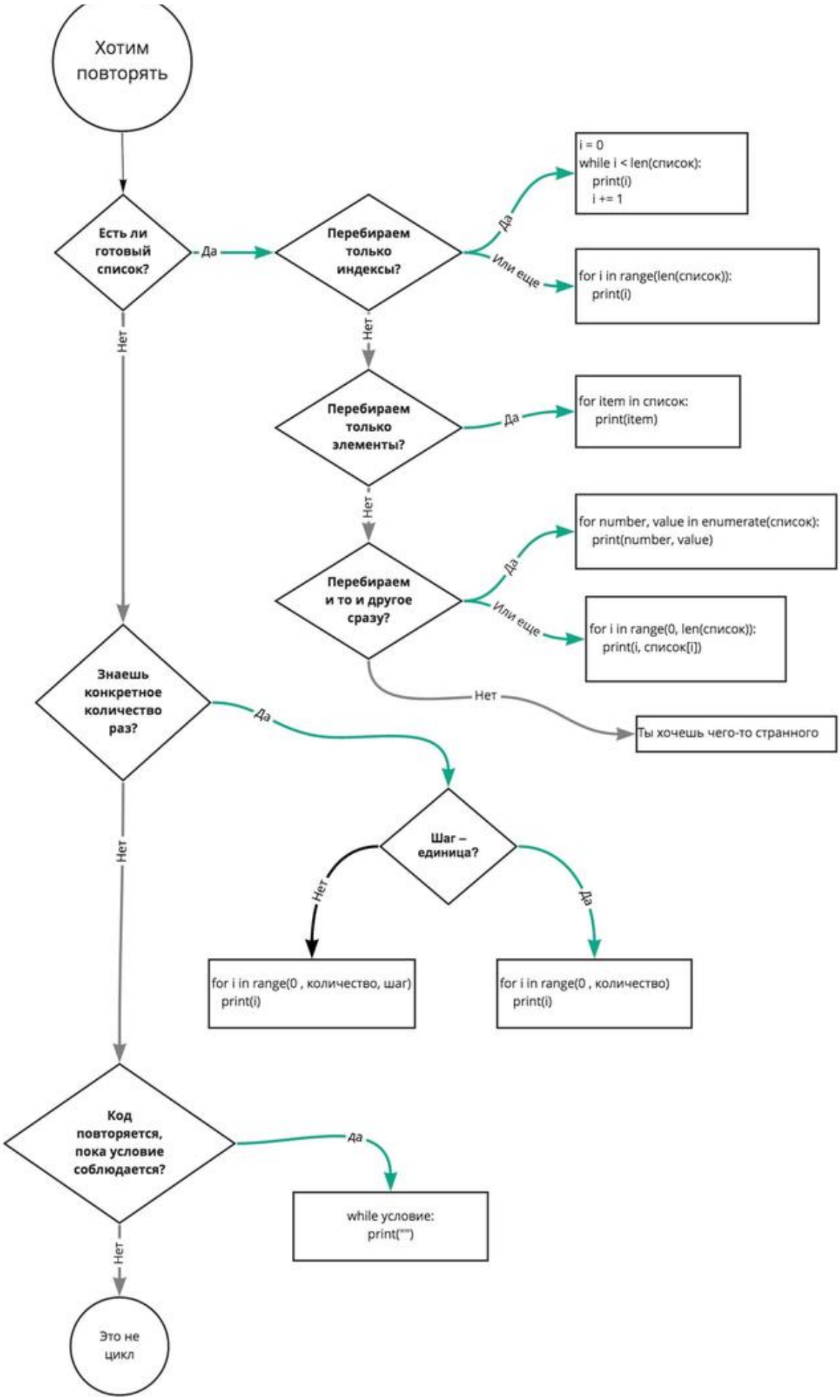
for i, letter in enumerate(letters, start=1):
    if i % 2 == 0:
        print(i, letter)
```

Выведет:

```
2 Bravo
4 Delta
6 Foxtrot
```

Как выбрать цикл

Предлагаем схему, которая поможет выбрать нужный цикл для решения задачи.


Хотим повторять
Есть ли готовый список?
Да
Нет
Перебираем только индексы?
Да
Или еще
Нет
i = 0
while i < len(список):
print(i)
i += 1
for i in range(len(список)):
print(i)
Перебираем только элементы?
Да
Нет
for item in список:
print(item)
Перебираем и то и другое сразу?
Да
Или еще
Нет
for number, value in enumerate(список):
print(number, value)
for i in range(0, len(список)):
print(i, список[i])
Ты хочешь чего-то странного
Знаешь конкретное количество раз?
Да
Нет
Шаг – единица?
Нет
Да
for i in range(0 , количество, шаг)
print(i)
for i in range(0 , количество)
print(i)
Код повторяется, пока условие соблюдается?
Да
Нет
while условие:
print("")
Это не цикл