Циклы в программировании.

Каждый язык программирования, с которым я сталкивался, содержит какую-нибудь **конструкцию цикла**. В большей части языков есть больше одной такой конструкции. В мире Python есть два типа циклов:

* Цикл **for**
* Цикл **while**

Я заметил, что **цикл for** популярнее второго. Циклы используются в тех случаях, когда нам нужно сделать что-нибудь много раз. Нередко вам придется выполнить какую-нибудь операцию (или ряд операций) в части данных снова и снова. Тут то и вступают в силу циклы. Благодаря им становится возможно максимально упростить данный вопрос. Давайте подробно разберём, как работают эти структуры!

[**Цикл for**](https://python-scripts.com/loops-for-while#for)

Как было сказано ранее, мы используем **цикл** в тех случаях, когда вам нужно повторить что-нибудь n-ное количество раз. Это проще понять, если взглянуть на пример. Мы используем встроенную [функцию Python](https://python-scripts.com/functions-python) range. [Функция range](https://python-scripts.com/range) создаст список длинной в «n» элементов. В Python версии 2.Х существует другая функция под названием **xrange**, которая является генератором чисел и не такая ресурсоемкая, как **range**. Ранее разработчики **сменили xrange на range** в Python 3. Вот пример:

|  |  |
| --- | --- |
| 1 | print(range(5)) # ответ: range(0, 5) |

Как вы видите, функция **range** взяла целое число и вернула **объект range**. Функция range также принимает начальное значение, конечное значение и значение шага. Вот еще два примера:

|  |  |
| --- | --- |
| 1  2  3  4  5 | a = range(5, 10)  print(a) # range(5, 10)    b = list(range(1, 10, 2))  print(b) # [1, 3, 5, 7, 9] |

В пером примере показано, что вы можете передать **начальное** и **конечное** значение, и функция range вернет числа, начиная с начального значения вплоть до (но не включая) последнее значение. Например, при запросе 5-10 мы получим 5-9. Во втором примере видно, как использовать [функцию списка](https://python-scripts.com/lists-tuples-dictionaries) (list) для того, чтобы **функция range** вернула каждый второй элемент, между 1 и 10. Так что она начинает с 1, пропускает 2 и так далее. Теперь вы, наверное, гадаете, что же именно она будет делать с **циклами**? Что-ж, есть один простой способ показать, как работает цикл с использованием **функции range**! Давайте взглянем:

|  |  |
| --- | --- |
| 1  2 | for number in range(5):      print(number) |

Что здесь произошло? Давайте почитаем слева на право, чтобы понять это. Для каждого числа в диапазоне 5 мы вводим число. Мы знаем, что если мы вызываем range со значением 5, мы получим список из 5 элементов. Так что каждый раз, проходя через цикл, она выводит каждый из элементов. [Цикл for](https://python-scripts.com/loops-for-while), показанный выше, может быть эквивалентом следующего:

|  |  |
| --- | --- |
| 1  2 | for number in [0, 1, 2, 3, 4]:      print(number) |

Функция **range** лишь делает результат несколько меньшим. **Цикл for** может обходить любой итератор Python. Давайте взглянем, может ли он выполнять итерацию со словарем.

|  |  |
| --- | --- |
| 1  2  3  4 | a\_dict = {"one":1, "two":2, "three":3}    for key in a\_dict:      print(key) |

Когда вы используете **for в словаре**, вы увидите, что он автоматически **перебирает ключи**. Вам не нужно указывать ключ for в **a\_dict.keys()** (впрочем, это также работает). Python делает только нужные нам вещи. Вы возможно думаете, почему ключи выводятся в другом порядке, отличном от того, какой был указан в словаре? Как мы знаем из [соответствующей статьи](https://python-scripts.com/lists-tuples-dictionaries), словари не упорядочены, так что мы можем выполнять итерацию над ними, при этом ключи могут быть в любом порядке. Теперь, зная, что ключи могут быть **отсортированы**, вы можете отсортировать их до итерации. Давайте немного изменим словарь, чтобы увидеть, как это работает.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | a\_dict = {1:"one", 2:"two", 3:"three"}  keys = a\_dict.keys()    keys = sorted(keys)  for key in keys:      print(key) |

Результат:

|  |  |
| --- | --- |
| 1  2  3 | 1  2  3 |

Давайте остановимся и разберемся с тем, что делает этот код. Во-первых, мы [создали словарь](https://python-scripts.com/lists-tuples-dictionaries), в котором ключи выступают в качестве целых чисел, вместо строк. Далее, мы **извлекли ключи из словаря**. Каждый раз, когда вы взываете метод **keys**(), он возвращает неупорядоченный список ключей. Если вы выведите их, и увидите, что они расположен в порядке по возрастанию, то это просто случайность. Теперь у нас есть доступ к ключам словаря, которые хранятся в переменной, под названием **keys**. Мы **сортируем наш список**, после чего используем [цикл for](https://python-scripts.com/loops-for-while) в нем. Теперь мы готовы к тому, чтобы сделать все немного интереснее. Мы попробуем применить **цикл в функции range**, но нам нужно вывести только целые числа. Чтобы сделать это, нам нужно использовать условный оператор вместо параметра шага range. Это можно сделать следующим образом:

|  |  |
| --- | --- |
| 1  2  3 | for number in range(10):      if number % 2 == 0:          print(number) |

Результат:

|  |  |
| --- | --- |
| 1  2  3  4  5 | 0  2  4  6  8 |

Вы наверное гадаете, что вообще здесь происходит? Что еще за знак процента? В Python, % называется **оператором модуля**. Когда вы используете оператор модуля, он **возвращает остаток**. Когда вы делите целое число на два, вы получаете число без остатка, так что мы выводим эти числа. Вам, возможно, не захочется использовать оператор модуля часто в будущем, но в моей работе он нередко помогает. Теперь мы можем взглянуть на [цикл while](https://python-scripts.com/loops-for-while).

[**Цикл while**](https://python-scripts.com/loops-for-while#while)

**Цикл while** также используется для **повторения частей кода**, но вместо зацикливания на n количество раз, он выполняет работу до тех пор, пока не **достигнет определенного условия**. Давайте взглянем на простой пример:

|  |  |
| --- | --- |
| 1  2  3  4 | i = 0  while i < 10:      print(i)      i = i + 1 |

**Цикл while** является своего рода условным оператором. Вот что значит этот код: пока переменная i меньше единицы, её нужно выводить на экран. Далее, в конце, мы увеличиваем её значение на единицу. Если вы запустите этот код, он выдаст от 0 до 9, каждая цифра будет в отдельной строке, после чего задача будет выполнена. Если вы удалите ту часть, в которой мы увеличиваем значение i, то мы получим **бесконечный цикл**. Как правило – это плохо. **Бесконечные циклы** известны как логические ошибки, и их нужно избегать. Существует другой способ вырваться из цикла, для этого нужно использовать встроенную функцию **break**. Давайте посмотрим, как это работает:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | while i < 10:      print(i)        if i == 5:          break        i += 1 |

В этой части кода мы добавили условное выражение для проверки того, равняется ли когда-либо переменная i цифре 5. Если нет, тогда мы **разрываем цикл**. Как вы видите в выдаче кода, как только значение достигает пяти, код останавливается, даже если мы ранее указали **while** продолжать цикл, пока переменная не достигнет значения 10. Обратите внимание на то, что мы изменили то, как мы увеличиваем значение при помощи **+=**. Это удобный ярлык, который вы можете также использовать в других операциях, таких как вычитание **-=** и умножение **\*=**. Встроенный **break** также известен как инструмент управления потока. Существует еще один, под названием **continue**, который в основном используется для пропуска итерации, или перейти к следующей итерации. Вот один из способов его применения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | i = 0    while i < 10:      if i == 3:          i += 1          continue        print(i)      if i == 5:          break        i += 1 |

Слегка запутанно, не так ли? Мы добавили второе условное выражение, которое проверяет, не равняется ли i трем. Если да, мы увеличиваем переменную и переходим к **следующему циклу**, который удачно пропускает вывод значения 3 на экран. Как и ранее, когда мы достигаем значения 5, мы **разрываем цикл**. Существует еще одна тема, касающаяся циклов, которую нам нужно затронуть – это **оператор else**.

**Зачем нужен else при работе с циклами?**

**Оператор else** в циклах выполняется только в том случае, если цикл выполнен успешно. Главная задача **оператора else**, это поиск объектов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | my\_list = [1, 2, 3, 4, 5]    for i in my\_list:      if i == 3:          print("Item found!")          break      print(i)  else:      print("Item not found!") |

В этом коде мы **разорвали цикл**, когда i равно 3. Это приводит к пропуску **оператора else**. Если вы хотите провести эксперимент, вы можете изменить условное выражение, чтобы посмотреть на значение, которое находится вне списка, и которое приведет оператор else к выполнению. Честно, ни разу не видел, чтобы кто-либо использовал данную структуру за все годы работы. Большая часть примеров, которые я видел, приведена блогерами, которые пытаются объяснить, как это работает. Я видел несколько людей, которые использовали эту структуру для провоцирования **ошибки**, когда объект не удается найти в искомом цикле.

**Подведем итоги**

Надеюсь, с этого момента вы осознали всю значимость [циклов в Python](https://python-scripts.com/loops-for-while). Они делают повторение очень простым, и весьма понятным. Вы будете сталкиваться с **циклом for** намного чаще, чем с **циклом while**. Если вы все еще не совсем понимаете, как это работает, настоятельно рекомендую перечитать эту статью, перед тем как продолжить.