Часть 1. Цикл while

Цикл **while** в языке программирования Python предназначен для выполнения набора команд до тех пор, пока указанное условие верно. Синтаксис:

```
while условие:
    инструкции
```

Блок инструкций цикла располагается со следующей строки и каждая инструкция должна иметь отступы от начала цикла. После окончания выполнения блока команд, относящихся к while, управление возвращается на строку с условием и, если оно выполнено, то выполнение блока команд повторяется, а если не выполнено, то продолжается выполнение команд, записанных после while.

С помощью while очень легко организовать вечный цикл, поэтому необходимо следить за тем, чтобы в блоке команд происходили изменения, которые приведут к тому, что в какой-то момент условие перестанет быть истинным.

Рассмотрим пример.

Есть число N. Необходимо вывести все числа по возрастанию от 1 до N. Для решения этой задачи нужно завести счётчик (переменную index), который будет равен текущему числу. Вначале это единица. Пока значение счетчика не превысит N, необходимо выводить его текущее значение и каждый раз увеличить его на единицу:

```
index = 5
while index <= 10 :
    # n = int(input())
    print(index, end=" ")
    index = index + 1
    if index % 4 ==0:
      break

```

```
5 6 7 8
```

Синтаксический сахар

Конструкции типа index = index + 1 часто используются в Python, поэтому создатели языка добавили сокращенный вариант: index += 1. Они отличаются только способом записи. Интерпретатор сам превратит сокращенную конструкцию в развернутую.

Термин "синтаксический сахар" в Python относится к конструкциям в языке, которые предоставляют более краткий, удобный и интуитивно понятный синтаксис для выполнения определенных задач. Эти конструкции не вносят новую функциональность в язык, но улучшают читаемость и удобство написания кода. Мы уже использовали их в предыдущих уроках.

Существуют сокращенные формы для всех арифметических операций:

- a = a + 1 → a += 1
- a = a - 1 → a -= 1
- a = a * 2 → a *= 2
- a = a / 1 → a /= 1

```
a = 3; b = 4;
a *= b + 10 # a = a * (b + 10)
print (a)
```

```
42
```

Сокращение двойного неравенства до вида `1 < x < 10` также можно отнести к подобным улучшениям.

Конструкции из концепции "синтаксического сахара" помогают нагляднее и лаконичнее работать со списками и словарями, генераторами или даже файлами. Мы увидим некоторые из этих конструкций в соответствующих темах.

Бесконечный цикл while

Это цикл, в котором условие никогда не становится ложным. Это значит, что тело исполняется снова и снова, а цикл никогда не заканчивается.

Приведем пример **бесконечного** цикла:

```
a = 1

while a == 1:
    b = input('Как тебя зовут?')
    print('Привет', b, ', Добро пожаловать')
```

Если запустить этот код, то программа войдет в бесконечный цикл и будет снова и снова спрашивать имена. Цикл не остановится до тех пор, пока пользователь не нажмёт Ctrl + C либо кнопку Stop.

Часть 2. Операторы break, continue и else

Для циклов while применимы те же операторы, что и для цикла for:

- **pass** (определяет пустой блок, который ничего не делает),
- **break** (позволяет выйти из цикла досрочно),
- **continue** (позволяет перейти к следующей итерации цикла без выполнения оставшихся инструкций),
- **else** (проверяет цикл на экстренный выход).

Оператор break

Инструкция для прерывания цикла называется break. После её выполнения работа цикла прекращается (как будто не было выполнено условие цикла). Осмысленное использование конструкции break возможно, только если выполнено какое-то условие, то есть break должен вызываться только внутри if (находящегося внутри цикла). Использование break — плохой тон в программировании и, по возможности, следует обходиться без него.

```
i = 1
while i < 6:
    print(i)
    if i == 3:
        break
    i += 1
```

```
1
2
3
```

В Python нет прямой конструкции для цикла с постусловием (do-while), как в некоторых других языках программирования. Однако мы можем имитировать цикл с постусловием с использованием обычного цикла while, дополнительного условия внутри цикла и оператора break. Например:

```
counter = 0
while True:
    print("Этот блок выполнится хотя бы один раз")
    counter += 1
    if counter >= 0:
        break
```

```
Этот блок выполнится хотя бы один раз
```

Оператор continue

Оператор continue позволяет начать следующий проход цикла, минуя оставшиеся инструкции:

```
i = 0
while i < 6:
    i += 1
    if i == 3:
        continue
    print(i)
```

```
1
2
```

```
4
5
6
```

Оператор pass

Pass — оператор-заглушка, равноценный отсутствию операции. В ходе исполнения данного оператора ничего не происходит, поэтому он может использоваться в тех местах, где это синтаксически необходимо.

Зачастую pass используется там, где код пока ещё не появился, но планируется. Кроме этого, иногда его используют при отладке, разместив на строчке с ним точку остановки.

```
counter = 11
while counter <= 10:
    pass # TODO: позже придумаем что хотим делать в цикле
```

Оператор else

В языке Python к циклу while можно написать блок **else**. В этом случае блок в else исполняется, когда условие цикла становится ложным (в отличие от цикла for, в котором для этой цели используется оператор break). Например:

```
a = 1

while a < 5:
    print('условие верно')
    a = a + 1
else:
    print('условие неверно, a =', a)
```

```
условие верно
условие верно
условие верно
условие верно
условие неверно, a = 5
```

Программа исполняет код цикла while до тех, пока условие истинно, то есть пока значение a меньше 5. Поскольку начальное значение a равно 1, а с каждым циклом оно увеличивается на 1, условие станет ложным, когда программа доберется до четвертой итерации — в этот момент значение a изменится с 4 до 5. Программа проверит условие еще раз, убедится, что оно ложно и исполнит блок else, отобразив «условие неверно».

Часть 3. Пример использования цикла while.

Задача: разбить число на отдельные цифры.

```
num = int(input())
while num > 0:
    print(num % 10)  # отделение последней цифры от числа
    num //= 10       # уменьшение числа на разряд
```

```
584
4
8
5
```

В данном случае условие цикла является выполнимым, поскольку последнее целочисленное деление // даст в результате 0.

Таким же образом делается подсчёт количества разрядов и другие подобные задачи.

Вывод

Цикл while в языках программирования (включая Python) предназначен для многократного выполнения блока кода до тех пор, пока определенное условие остается истинным. Он особенно полезен, когда количество итераций заранее неизвестно, и цикл должен выполняться до выполнения определенного условия.

Вот несколько основных сценариев применения цикла while:

- решение задач с неизвестным количеством итераций (итерация до достижения заданного условия);
- интерактивное взаимодействие с пользователем (ввод данных с командной строки в бесконечном цикле);
- создание анимации;
- работа с данными в реальном времени (для постоянного чтения и обновления данных).

Важно помнить, что использование цикла while требует осторожности, чтобы избежать неконтролируемых бесконечных циклов. Необходимо уделять внимание условию завершения цикла и обновлению переменных внутри цикла, чтобы он выполнился корректно и не привел к нежелательным результатам (например, к зависанию программы или к повышенному потреблению ресурсов компьютера).

Часть 4. * Оценка сложности алгоритма в нотации O(n)

[продвинутая необязательная тема]

Поскольку изученные циклы и условные операторы уже позволяют реализовывать довольно сложные алгоритмы, стоит поговорить о вычислительной сложности программ (алгоритмов).

Существует несколько способов измерения сложности алгоритма. Основным критерием является скорость алгоритма, но не менее важны и другие показатели – требования к объёму оперативной памяти и свободному месте на диске. Например, использование быстрого алгоритма не приведёт к ожидаемым результатам, если для его работы понадобится больше памяти, чем есть у компьютера. Однако в этой теме сосредоточимся всё же на оценке времени работы алгоритма — как на наиболее часто используемом критерии.

Самыми сложными частями программы обычно является выполнение циклов и вызов процедур, поэтому они чаще всего подвергаются оценке потенциального времени исполнения.

Допустим, нам надо оценить сложность какого-то алгоритма (например, состоящего из двух вложенных циклов). Понятно, что чем сложнее алгоритм, тем больше он выполняет операций и, соответственно, выполняется дольше по времени. Но ещё больше время выполнения алгоритма зависит от количества входных данных. Обработка одного файла и обработка миллиона таких же файлов может занимать абсолютно разное время.

Как же нам оценить абстактную величину сложности вне зависимости от входных данных?

Для этого в программировании используется так называемая нотации "О большое" (Big O notation), которая оценивает скорость работы алгоритма по мере увеличения входных данных (до потенциальной бесконечности). Если объяснять самыми простыми словами, то "О большое" это оценка самого худшего случая из всех возможных. Оценка худшего случая делает алгоритмы предсказуемыми, поскольку игнорирует разнообразные "удобные данные". Это означает, что программисты могут быть уверены в том, как поведет себя алгоритм в самых неблагоприятных условиях.

Чтобы не приводить сложную математику об асимптотике функций, просто рассмотрим несколько примеров. Пусть **n** — количество входных данных (которое растет, например, от 1 до 1 000 000 и далее).

Линейная сложность O(n)
O(n) означает, что время выполнения алгоритма прямо пропорционально размеру входных данных. Чем больше данных, тем больше времени потребуется.
Такую сложность имеют как раз циклы for и while.

```
for i in range(1, n):
    print(i)
```

В данном случае если n = 100, то чтобы вывести на экран число, нам потребуется 100 шагов цикла. При увеличении n в тысячу раз нам потребуется сделать ровно в 1000 раз больше шагов. Если n = 1_000_000_000, то нам потребуется один миллиард шагов. И так далее, мы можем брать сколь угодно большую цифру, ничего не изменится.
То же самое будет и с циклом while:

```
while i < n:
    print(i)
    i += 1
```

Таким образом мы можем сказать, что алгоритм, состоящий из одного цикла и зависящий от числа элементов n, имеет сложность O(n).

Константная сложность O(1)
O(1) означает, что время выполнения алгоритма не зависит от размера входных данных и всегда остается постоянным.
Примером такого алгоритма является простая арифметическая операция сложения или вычитания.

```
n += 1
```

Мы видим, что каким бы большим или маленьким ни было n, данная операция всё равно выполнится за один шаг. То есть сложность независима и постоянна.
Это самая лучшая сложность работы алгоритма, к которой в идеале следует стремиться. Однако на практике такая сложность достигается лишь в очень малом количестве алгоритмов (операций).

Квадратичная сложность O(n*n)
O(n*n) означает, что время выполнения алгоритма пропорционально квадрату размера входных данных.
Рассмотрим следующий код, состоящий из двух вложенных циклов:

```
for i in range(1, n):
    for j in range(1, n):
        print(i, j)
```

Чтобы вывести на экран все пары точек, потребуется n раз сделать цикл, так же состоящий из n итераций. Получается, чтобы напечатать на экран все пары элементов как раз потребуется n*n шагов.
На примере этого алгоритма можно посмотреть, что значит "худший случай". Рассмотрим такую модификацию:

```
for i in range(1, n):
    for j in range(i, n):
        print(i, j)
```

Казалось бы, этот алгоритм явно теперь делает меньше шагов, чем предыдущий. Мы даже можем посчитать, сколько: n * (n-1) / 2 .
Упростив выражение, получим

```
0.5 * (n * n - n)
```

А теперь представим, что n ОЧЕНЬ большое. При этом число n * n будет просто в огромное количество раз больше самого n (которое и так большое). Тогда получается, что число n в этой разности в скобках уже почти никак не влияет на произведение n * n, которое куда более значимо. Поэтому в нотации "О большое" существует правило отбрасывания незначимых чисел в перспективе больших значений, а также правило отбрасывания констант (по той же причине).
В итоге, в худшем случае (числа очень большие) при отбрасывании незначимых чисел и констант у нас всё равно остаётся сложность n * n — как и у первоначальной версии алгоритма.

Факториальная сложность O(n!)
O(n!) означает, что время выполнения алгоритма растет факториально с размером входных данных.
Примером может быть алгоритм полного перебора всех перестановок элементов, который в случае n = 3 даёт 3(3-1)(3-2) = 3! = 6 разных вариантов: `(1 2 3, 1 3 2, 2 1 3, 2 3 1, 3 1 2, 3 2 1)`. Однако в случае n = 8 вариантов уже 40320.

Существуют и **другие** оценки алгоритмов. Это O(log n), O(n log n) и O(2^n). Коснемся их, когда будем проходить соответствующие темы.

Получается, что алгоритм сложности O(n) намного эффективнее алгоритма O(n*n). Поэтому если задачу получается решить двумя разными способами, стоит оценить оба алгоритма в нотации "О большое" и выбрать тот, который быстрее.

Итак, знание нотации "О большое" крайне важно, поскольку оно позволяет программистам сравнивать алгоритмы и выбирать наиболее эффективные из них для конкретных задач.