**Условные конструкции**

**Алгоритм** — набор инструкций, описывающих порядок действий исполнителя для достижения результата решения задачи за конечное число действий. В старой трактовке вместо слова «порядок» использовалось слово «последовательность», но по мере развития параллельности в работе компьютеров слово «последовательность» стали заменять более общим словом «порядок». Это связано с тем, что работа каких-то инструкций алгоритма может быть зависима от других инструкций или результатов их работы. Таким образом, некоторые инструкции должны выполняться строго после завершения работы инструкций, от которых они зависят. Независимые инструкции или инструкции, ставшие независимыми из-за завершения работы инструкций, от которых они зависят, могут выполняться в произвольном порядке, параллельно или одновременно, если это позволяют используемые процессор и операционная система.

Часто в качестве исполнителя выступает некоторый механизм (компьютер, токарный станок, швейная машина), но понятие алгоритма необязательно относится к компьютерным программам, так, например, чётко описанный рецепт приготовления блюда также является алгоритмом, в таком случае исполнителем является человек.

**Формальные свойства алгоритмов:**

* **Дискретность** — алгоритм должен представлять процесс решения задачи как последовательное выполнение некоторых простых шагов. При этом для выполнения каждого шага алгоритма требуется конечный отрезок времени, то есть преобразование исходных данных в результат осуществляется во времени дискретно.
* **Детерминированность** (определённость). В каждый момент времени следующий шаг работы однозначно определяется состоянием системы. Таким образом, алгоритм выдаёт один и тот же результат (ответ) для одних и тех же исходных данных. С другой стороны, существуют вероятностные алгоритмы, в которых следующий шаг работы зависит от текущего состояния системы и генерируемого случайного числа. Однако при включении метода генерации случайных чисел в список «исходных данных» вероятностный алгоритм становится подвидом обычного.
* **Понятность** — алгоритм должен включать только те команды, которые доступны исполнителю и входят в его систему команд.
* **Завершаемость** (конечность) — при корректно заданных исходных данных алгоритм должен завершать работу и выдавать результат за конечное число шагов.
* **Массовость** (универсальность). Алгоритм должен быть применим к разным наборам исходных данных.
* **Результативность** — завершение алгоритма определёнными результатами. Алгоритм содержит ошибки, если приводит к получению неправильных результатов либо не даёт результатов вовсе. Алгоритм не содержит ошибок, если он даёт правильные результаты для любых допустимых исходных данных.

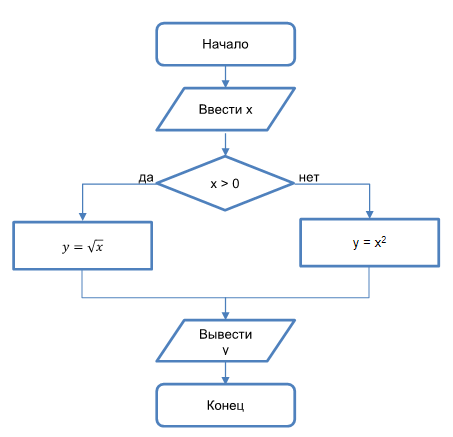
**Виды простейших алгоритмов:**

* **Линейный алгоритм** — набор команд (указаний), выполняемых последовательно во времени друг за другом.
* **Разветвляющийся алгоритм** — алгоритм, содержащий хотя бы одно условие, в результате проверки которого может осуществляться разделение на несколько параллельных ветвей алгоритма.
* **Циклический алгоритм** — алгоритм, предусматривающий многократное повторение одного и того же действия (одних и тех же операций) над новыми исходными данными. К циклическим алгоритмам сводится большинство методов вычислений, перебора вариантов. Цикл программы — последовательность команд (серия, тело цикла), которая может выполняться многократно (для новых исходных данных) до удовлетворения некоторого условия.

**Линейные алгоритмы** состоят из последовательного набора команд, которые необходимо выполнить одна за другой. Программы, реализующие именно такие алгоритмы, создавались в предыдущих работах.

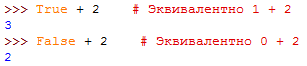
**Алгоритм с ветвлениями** содержит условия, в зависимости от которых выполняется одна или другая последовательность действий. Каждая из них, в свою очередь, также может (но не обязана) содержать ветвления.

Пример алгоритма с ветвлением



**Логические выражения**

Условные операторы позволяют в зависимости от значения логического выражения выполнить отдельный участок программы или, наоборот, не выполнять его. Логические выражения возвращают только два значения **True** (истина) или **False** (ложь), которые ведут себя как целые числа 1 и 0 соответственно:



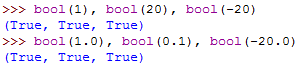
Логическое значение можно сохранить в переменной:



Любой объект в логическом контексте может интерпретироваться как истина (**True**) или как ложь (**False**). Для определения логического значения можно использовать функцию **bool()**.

Значение **True** возвращают следующие объекты:

* любое число, не равное нулю:



* не пустой объект:



Следующие объекты интерпретируются как **False**:

* число, равное нулю:



* пустой объект:



* значение **None**:



**Операторы сравнения**

Операторы сравнения используются в логических выражениях. Перечислим их:

* **==** равно:



* **!=** не равно:



* **<** меньше:



* **>** больше:



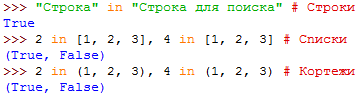
* **<=** меньше или равно:



* **>=** больше или равно:



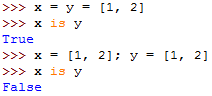
* **in** проверка на вхождение в последовательность:



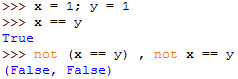
Оператор **in** можно также использовать для проверки существования ключа словаря:



* **is** проверяет, ссылаются ли две переменные на один и тот же объект. Если переменные ссылаются на один и тот же объект, то оператор **is** возвращает значение **True**:



Значение логического выражения можно инвертировать с помощью оператора **not**:



Если переменные х и у равны, то возвращается значение **Тrue**, но т. к. перед выражением стоит оператор **not**, выражение вернет **Fa1se**. Круглые скобки можно не указывать, т. к. оператор **not** имеет более низкий приоритет выполнения, чем операторы сравнения.

При необходимости инвертировать значение оператора **in**, оператор **not** указывается непосредственно перед этим оператором:



Чтобы инвертировать значение оператора **is**, оператор **not** указывается непосредственно после этого оператора:

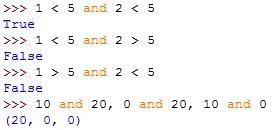


В логическом выражении можно указывать сразу несколько условий:

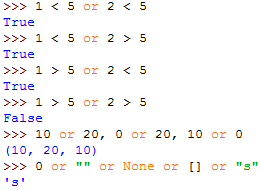


Несколько логических выражений можно объединить в одно большое с помощью следующих операторов:

* **and** логическое И. **х and у** если х интерпретируется как **Fa1se**, то возвращается х, в противном случае - у:



* **or** **логическое** ИЛИ. **х or у** если х интерпретируется как **Fa1se**, то возвращается у, в противном случае - х:



Это выражение вернет **True** только в случае, если оба выражения вернут **True**:

**х1 == х2 and х2 != х3**

А это выражение вернет **True**, если хотя бы одно из выражений вернет **True**:

**х1 == х2 or х3 == х4**

Перечислим операторы сравнения в порядке убывания приоритета:

1. **<, >, <=, >=, ==, !=, is, is not, in, not in**.
2. **not** логическое отрицание.
3. **and** логическое И.
4. **or** логическое ИЛИ.

**Оператор if (if-else)**

В **Python** для реализации ветвлений используется оператор **if**. Самой простой его формой является

**if условие:**

**операторы**

где ***условие*** - это логическое выражение, а ***операторы*** - это последовательность каких-либо других команд.

Пример:

**if х > 5:**

**print('х больше пяти')**

Блок операторов не может быть пустым. Если необходимо создать блок, который ничего не делает (например, чтобы набросать структуру кода, а затем реализовывать отдельные его части, или в каких-нибудь других целях), то используется специальный оператор ***pass****.*

Оператор ***pass*** не выполняет никаких действий и обычно используется, когда нужно создать логически пустой блок кода там, где синтаксис языка требует наличия каких- либо операторов. Пример:

**if value is None:**

**pass # исправить позже**

Очень важны в коде на **Python** отступы, так как именно по ним интерпретатор определяет границы блоков кода и уровни их вложенности. Основным правилом является то, что операторы, находящиеся внутри одного блока, должны выделяться слева одинаковым количеством пробелов или знаков табуляции, но хорошим стилем является внутри одного файла выделять каждый блок одинаковым образом, символами пробела или табуляции в количестве, кратным какому-либо определённому числу, и очень нежелательно смешивать пробелы и знаки табуляции. Согласно соглашениям написания кода, на **Python** (PEP 8), принято выделять каждый уровень вложенности **четырьмя** пробелами.

Оператор **if** также можно использовать в однострочной форме:

**if условие: операторы**

Если блок операторов здесь состоит больше, чем из одной инструкции, то они разделяются точкой с запятой («;»). Однако такая форма нежелательна для использования, так как значительно снижает читабельность кода, и допустима только если блок операторов состоит из одной инструкции, и как она, так и условие, достаточно короткие. Но даже в этом случае предпочтительнее всё же размещать её на новой строке.

Рассмотренная выше форма оператора **if** выполняет или не выполняет ветку кода в зависимости от истинности заданного выражения. Однако, как было сказано выше, разветвляющиеся алгоритмы характерны тем, что в зависимости от значения логического выражения может быть выполнена одна из двух веток кода. Такая конструкция в **Python** реализуется оператором **if-else:**

**if условие:**

**блок операторов 1**

**else :**

**блок операторов 2**

Если условие истинно, то выполняются операторы из первого блока, иначе - второго.

В качестве примера рассмотрим реализацию алгоритма из блок-схемы приведенной выше:

**х = int (input('х = '))**

**if х > 0:**

**у = х \*\* 0.5**

**else :**

**у = х \*\* 2**

**print(у)**

Так как блоки операторов внутри **if** могут содержать любые операторы и их последовательности, очевидно, что операторы **if** могут быть вложенными. Пример:

**if 0 < x < 7:**

**print('Значение x входит в заданный диапазон, продолжаем')**

**y = 2 \* x - 5**

**if y < 0:**

**print('Значение y отрицательно')**

**else:**

**if y > 0:**

**print('Значение y положительно')**

**else:**

**print('y = 0')**

**Оператор if-elif-else**

В последнем примере, во втором вложенном **if** мы видим ситуацию, в которой мы поочерёдно проверяем несколько условий (в данном случае, отрицателен y, положителен или равен нулю). В общем случае это можно записать так:

**if условие1:**

**команды1**

**else:**

**if условие2:**

**команды2**

**else:**

**if условие3:**

**команды3**

**...**

**else:**

**if условиеN:**

**командыN**

**else:**

**командыM**

Подобный приём называется каскадированием условных операторов. Ситуации, в которых необходимо выполнить подобную проверку, встречаются очень часто, однако видно, что каскад условных операторов выглядит несколько нечитабельно и легко запутаться при его написании. В разных языках программирования есть разные подходы к решению этой проблемы.

Для языков, унаследовавших синтаксис от C (C, C++, Java, C# и т.д.) характерен оператор переключения **switch** (в не C-подобных языках, которые также имеют этот оператор, он может называться **case** или ещё как-нибудь иначе). В зависимости от значения выражения он выполняет одну из веток кода. Его недостатком является то, что в большинстве языков, где он реализован (исключения – ***C#*, JavaScript**) управляющее выражение может иметь только целый тип, а условия переключения могут проверять его значение только на равенство определённым константам. В большинстве случаев этого достаточно, но в других приходится использовать каскад операторов **if**.

В функциональных языках программирования, таких как **Haskell** или **F#**, есть довольно мощный механизм сопоставления с образцом.

В **Python** используется крайне простое и, тем не менее, гибкое и удобное решение этой проблемы: оператор ветвления с несколькими условиями.

**if условие1:**

**операторы1**

**elif условие2:**

**операторы2**

**elif условие3:**

**операторы3**

**...**

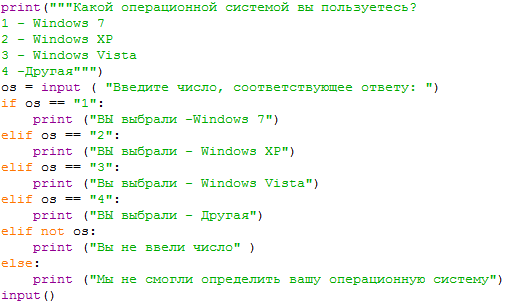
**else:**

**операторыN**

Как и в обычной форме оператора **if**, ветка **else** здесь необязательна.

Таким образом, в общем виде оператор **if** в **Python** выглядит так: сначала идёт ключевое слово **if**, условие, двоеточие и блок кода, затем может идти любое количество необязательных блоков **elif**, в конце может указываться необязательный блок **else**. Сначала интерпретатор проверяет истинность условия в блоке **if**, и если оно истинно, то выполняет соответствующие команды и переходит к коду после всего оператора **if**, иначе, если присутствуют блоки **elif**, проверяет истинность соответствующих условий в каждом блоке по очереди, пока не найдёт истинное, после чего выполняет соответствующие команды и завершает выполнение оператора **if**. Если же все условия в блоках **if** и **elif** были ложными, то при наличии секции **else**, выполняются операторы в ней.

Пример



С помощью инструкции **elif** мы можем определить выбранное значение и вывести соответствующее сообщение. Обратите внимание на то, что логическое выражение не содержит операторов сравнения:

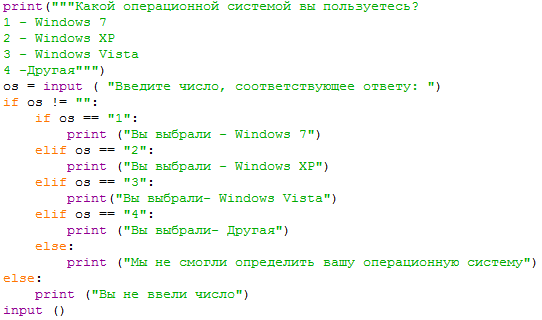
**elif not os:**

Такая запись эквивалентна следующей:

**elif os == "":**

Проверка на равенство выражения значению **True** выполняется по умолчанию. Так как пустая строка интерпретируется как **False**, мы инвертируем возвращаемое значение с помощью оператора **not**.

Один условный оператор можно вложить в другой. В этом случае отступ вложенной инструкции должен быть в два раза больше.



**if в виде выражения (условное выражение)**

Довольно часто возникает ситуация, когда нам нужно использовать то или иное выражение в зависимости от определённого условия. Поэтому существует такая конструкция, как условное выражение (или, иначе, «тернарный оператор»).

Синтаксис условного выражения в Python:

**выражение1 if условие else выражение2**

Результатом всего этого выражения является значение первого выражения, если условие истинно, или второго, если ложно.

Пример. Вместо

**if is\_ready:**

**state\_msg = 'Ready'**

**else:**

**state\_msg = 'Not ready yet'**

можно записать

**state\_msg = 'Ready' if is\_ready else 'Not ready yet'**