**Логический тип данных и операции**

**Логический тип данных**

Кроме уже известных нам целочисленных и строковых типов данных в Питоне существует также логический тип данных, который может принимать значения "истина" (True) или "ложь" (False).

По аналогии с арифметическими выражениями существуют логические выражения, которые могут быть истинными или ложными. Простое логическое выражение имеет вид <арифметическое выражение> <знак сравнения> <арифметическое выражение>. Например, если у нас есть переменные x и y с какими-то значениями, то логическое выражение x + y < 3 \* y в качестве первого арифметического выражения имеет x + y, в качестве знака сравнения < (меньше), а второе арифметическое выражение в нём 3 \* y.

В логических выражениях допустимы следующие знаки сравнений:

|  |  |
| --- | --- |
| Знак сравнения | Описание |
| < | меньше |
| > | больше |
| <= | меньше либо равно |
| >= | больше либо равно |
| == | равно |
| != | не равно |

В Питоне допустимы и логические выражения, содержащие несколько знаков сравнения, например x < y < z. При этом все сравнения обладают одинаковым приоритетом, который меньше, чем у любой арифметической операции.

Результат вычисления логического выражения можно сохранять в переменную, которая будет иметь тип bool. Переменные такого типа, как и числа и строки, являются неизменяемыми объектами.

Строки также могут сравниваться между собой. При этом сравнение происходит в лексикографическом порядке (как упорядочены слова в словаре).

**Логические операции**

Чтобы записать сложное логическое выражение, часто бывает необходимо воспользоваться логическими связками "и", "или" и "не". В Питоне они обозначаются как and, or и not соответственно. Операции and и or являеются бинарными, т.е. должны быть записаны между операндами, например x < 3 or y > 2. Операция not - унарная и должна быть записана перед единственным своим операндом.

Все логические операции имеют приоритет ниже, чем операции сравнения (а значит, и ниже чем арифметические операции). Среди логических операций наивысший приоритет имеет операция not, затем идет and и наименьший приоритет имеет операция or. На порядок выполнения опера

**Примеры использования логических выражений**

Одним из примеров использования логического выражения является проверка на делимость. Например, чтобы проверить, является ли число четным, необходимо сравнить остаток от деления этого числа на два с нулём:

|  |  |
| --- | --- |
| 1 | isEven = number % 2 == 0 |

Рассмотрим задачу о пересечении двух длительных по времени событий. Оба события характеризуются двумя числами - годами начала и конца. Необходимо определить, пересекались ли события во времени, при этом если одно событие началось в тот год, когда закончилось другое - они считаются пересекающимися.

Первая идея заключается в том, чтобы рассмотреть все возможные варианты расположения событий и выделить следующий критерий пересечения: если начало или конец одного из событий лежит между началом и концом другого, то они пересекаются. В виде программы это можно записать так:

|  |  |
| --- | --- |
| 1 | is1startIn2 = start2 <= start1 <= finish2 |
| 2 | is1finisIn2 = start2 <= finish1 <= finish2 |
| 3 | is1in2 = is1startIn2 or is1finishIn2 |
| 4 | is2startIn1 = start1 <= start2 <= finish1 |
| 5 | is2finisIn1 = start1 <= finish2 <= finish1 |
| 6 | is2in1 = is2startIn1 or is2finishIn1 |
| 7 | answer = is1in2 or is2in1 |

Если немного подумать, то можно придумать более короткий критерий для проверки такого пересечения: необходимо, чтобы начало первого события происходило не позже конца второго и начало второго события происходило раньше конца первого.

|  |  |
| --- | --- |
| 1 | answer = start1 <= finish2 and start2 <= finish1 |

Такой способ значительно проще.

Последнее изменение: Sunday, 2 September 2018, 17:29

**Условный оператор**

Наиболее частое применение логические выражения находят в условных операторах.

Условный оператор позволяет выполнять действия в зависимости от того, выполнено условие или нет. Записывается условный оператор как ''if <логическое выражение>:'', далее следует блок команд, который будет выполнен только если логическое выражение приняло значение True. Блок команд, который будет выполняться, выделяется отступами в 4 пробела (в IDE можно нажимать клавишу tab).

Рассмотрим, например, задачу о нахождении модуля числа. Если число отрицательное, то необходимо заменить его на минус это число. Решение выглядит так:

|  |  |
| --- | --- |
| 1 | x = int(input()) |
| 2 | if x < 0: |
| 3 | x = -x |
| 4 | print(x) |

В этой программе с отступом записана только одна строка, x = -x. При необходимости выполнить несколько команд все они должны быть записаны с тем же отступом. Команда print записана без отступа, поэтому она будет выполняться в любом случае, независимо от того, было ли условие в if'е истинным или нет.

В дополнение к if можно использовать оператор else: (иначе). Блок команд, который следует после него, будет выполняться если условие было ложным. Например, ту же задачу о выводе модуля числа можно было решить, не меняя значения переменной x:

|  |  |
| --- | --- |
| 1 | x = int(input()) |
| 2 | if x >= 0: |
| 3 | print(x) |
| 4 | else: |
| 5 | print(-x) |

Все команды, которые выполняются в блоке else, должны быть также записаны с отступом. Else должен следовать сразу за блоком команд if, без промежуточных команд, выполняемых безусловно. Else без соответствующего if'а не имеет смысла.

Если после if записано не логическое выражение, то оно будет приведено к логическому, как если бы от него была вызвана функция bool. Однако, злоупотреблять этим не следует, т.к. это ухудшает читаемость кода.

Для подсчета модуля числа в Питоне существует функция abs, которая избавляет от необходимости каждый раз писать подсчет модуля вручную.

В Питоне, как и во многих других языках программирования, если результат вычисления выражения однозначно понятен по уже вычисленной части, то оставшаяся часть выражения даже не считается. Например, выражение True or 5 // 0 == 42, не будет вызывать ошибки деления на ноль, т.к. по левой части выражения (True) уже понятно, что результат его вычисления также будет True и арифметическое выражение в правой части даже не будет вычисляться.

**Вложенный условный оператор и "иначе-если"**

**Вложенный условный оператор**

Внутри блока команд могут находиться другие условные операторы. Посмотрим сразу на примере. По заданному количеству глаз и ног нужно научиться отличать кошку, паука, морского гребешка и жучка. У морского гребешка бывает более сотни глаз, а у пауков их восемь. Также у пауков восемь ног, а у морского гребешка их нет совсем. У кошки четыре ноги, а у жучка - шесть ног, но глаз у обоих по два. Решение:

|  |  |
| --- | --- |
| 1 | eyes = int(input()) |
| 2 | legs = int(input()) |
| 3 | if eyes >= 8: |
| 4 | if legs == 8: |
| 5 | print("spider") |
| 6 | else: |
| 7 | print("scallop") |
| 8 | else: |
| 9 | if legs == 6: |
| 10 | print("bug") |
| 11 | else: |
| 12 | print("cat") |

Если вложенных условных операторов несколько, то, к какому из них относится else, можно понять по отступу. Отступ у else должен быть такой же, как у if, к которому он относится.

**Конструкция ''иначе-если''**

В некоторых ситуациях необходимо осуществить выбор больше чем из двух вариантов, которые могут быть обработаны с помощью if-else. Рассмотрим пример: необходимо вывести словом название числа 1 или 2 или сообщить, что это другое число:

|  |  |
| --- | --- |
| 1 | number = int(input()) |
| 2 | if number == 1: |
| 3 | print('One') |
| 4 | elif number == 2: |
| 5 | print('Two') |
| 6 | else: |
| 7 | print('Other') |

Здесь используется специальная конструкция elif, обозначающая "иначе, если", после которой записывается условие. Такая конструкция введена в язык Питон, потому что запись if-else приведет к увеличению отступа и ухудшению читаемости.

Конструкций elif может быть несколько, условия проверяются последовательно. Как только условие выполнено~--- запускается соответствующий этому условию блок команд и дальнейшая проверка не выполняется. Блок else является необязательным, как и в обычном if.

**Цикл while**

While переводится как "пока" и позволяет выполнять команды, до тех пор, пока условие верно. 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 несколько примеров.

Есть число N. Необходимо вывести все числа по возрастанию от 1 до N. Для решения этой задачи нужно завести счётчик (переменную i), 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:

|  |  |
| --- | --- |
| 1 | n = int(input()) |
| 2 | i = 1 |
| 3 | while i <= n: |
| 4 | print(i) |
| 5 | i = i + 1 |

Еще одна часто встречающаяся задача - поиск минимума (или максимума) в последовательности чисел. Пусть задана последовательность чисел, оканчивающаяся нулём. Необходимо найти минимальное число в этой последовательности. Эта задача может быть решена человеком: каждый раз когда ему называют очередное число, он сравнивает его с текущим запомненным минимумом и, при необходимости, запоминает новое минимальное число. В качестве первого запомненного числа нужно взять первый элемент последовательности, который должен быть считан отдельно до цикла.

|  |  |
| --- | --- |
| 1 | now = int(input()) |
| 2 | nowMin = now |
| 3 | while now != 0: |
| 4 | if now < nowMin: |
| 5 | nowMin = now |
| 6 | now = int(input()) |
| 7 | print(nowMin) |

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, по возможности, следует обходиться без него. Рассмотрим пример вечного цикла, выход из которого осуществляется с помощью break. Для этого решим задачу о выводе всех целых чисел от 1 до 100. Использовать break таким образом ни в коем случае не нужно, это просто пример:

|  |  |
| --- | --- |
| 1 | i = 1 |
| 2 | while True: |
| 3 | print(i) |
| 4 | i = i + 1 |
| 5 | if i > 100: |
| 6 | break |

В языке Питон к циклу while можно написать блок else. Команды в этом блоке будут выполняться, если цикл завершил свою работу нормальным образом (т.е. условие в какой-то момент перестало быть истинным) и не будут выполняться только в случае, если выход из цикла произошел с помощью команды break.

**Подсчет суммы и оператор continue**

Часто возникает задача о подсчете суммы последовательности. Для подсчета суммы чисел необходимо завести переменную, которая будет хранить накопленную на данный момент сумму и, при чтении очередного числа, прибавлять его к накопленной сумме:

|  |  |
| --- | --- |
| 1 | now = int(input()) |
| 2 | seqSum = 0 |
| 3 | while now != 0: |
| 4 | seqSum = seqSum + now |
| 5 | now = int(input()) |
| 6 | print(seqSum) |

Команда continue начинает исполнение тела цикла заново, начиная с проверки условия. Её нужно использовать, если начиная с какого-то места в теле цикла и при выполнении каких-то условий дальнейшие действия нежелательны.

Приведём пример использования continue (хотя при решении этой задачи можно и нужно обходиться без него): дана последовательность чисел, оканчивающаяся нулём. Необходимо вывести все положительные числа из этой последовательности. Решение:

|  |  |
| --- | --- |
| 1 | now = -1 |
| 2 | while now != 0: |
| 3 | now = int(input()) |
| 4 | if now <= 0: |
| 5 | continue |
| 6 | print(now) |

В этом решении есть интересный момент: перед циклом переменная инициализируется заведомо подходящим значением. Команда вывода будет выполняться только в том случае, если не выполнится условие в if