**Как устроены вещественные числа**

В отличие от целых чисел, вещественные числа в языке Питон имеют ограниченную длину.

Подумаем, как хранить десятичную дробь в памяти. Поскольку вещественных чисел бесконечно много (даже больше, чем натуральных), то нам придется ограничить точность. Например, мы можем хранить только несколько первых значащих цифр, не храня незначащие нули. Будем отдельно хранить целое число с первыми значащами цифрами и отдельно хранить степень числа 10, на которую нужно умножить это число.

Например, число 5.972\*10\*\*24 (это масса Земли в килограммах) можно сохранить как 5972 (цифры числа, мантисса) и 21 (на какую степень 10 нужно умножить число, экспонента). С помощью такого представления можно хранить вещественные числа любой размерности.

Примерно так и хранятся числа в памяти компьютера, однако вместо десятичной системы используется двоичные. На большинстве аппаратных систем в языке Питон для хранения float используется 64 бита, из которых 1 бит уходит на знак, 52 бита - на мантиссу и 11 бит - на экспоненту. Это не совсем правда, но достаточно неплохо описывает реальность.

52 бита дают около 15-16 десятичных знаков, которые будут хранится точно. 11 бит на экспоненту также накладывает ограничения на размерность хранимых чисел (примерно от -1000 до 1000 степени числа 10).

Любое вещественное число на языке Питон представимо в виде дроби, где в числителе хранится целое число, а в знаменателе находится какая-либо степень двойки. Например, 0.125 представимо как 1/8, а 0.1 как 3602879701896397/36028797018963968. Несложно заметить, что эта дробь не равно 0.1, т.е. хранение числа 0.1 точно в типе float невозможно, как и многих других "красивых" десятичных дробей.

В целом будет полезно представлять себе вещественное число X как отрезок [X - epsilon; X + epsilon]. Как же определить величину epsilon?

Для этого нужно понять, что погрешность не является абсолютной, т.е. одинаковой для всех чисел, а является относительной. Упрощенно, аппаратную погрешность хранения числа X можно оценить как X\*2\*\*(-54).

Чаще всего в задачах входные данные имеют определенную точность. Рассмотрим на примере: заданы два числа X и Y с точностью 6 знаков после точки (значит epsilon=5\*10\*\*(-7)) и по модулю не превосходящие 10\*\*9. Оценить абсолютную погрешность вычисления X \* Y. Рассмотрим худший случай, когда X и Y равны 10\*\*9 и отклонились на максимально возможное значение epsilon в одну сторону. Тогда результат вычисления будет выглядеть так:

(X + epsilon) \* (Y + epsilon) = XY + (X + Y) \* epsilon + epsilon\*\*2

Величина epsilon\*\*2 пренебрежимо мала, XY - это правильный ответ, а (X + Y) \* epsilon - искомое значение абсолютной погрешности. Подставим числа и получим:

2 \* 10\*\*9 \* 5 \* 10\*\*(-7) = 10\*\*3

Абсолютная погрешность вычисления составила 1000 (одну тысячу). Что довольно неожиданно и грустно.

Таким образом, становится понятно, что нужно аккуратно вычислять значение погрешности для сравения вещественных чисел.

**Основы работы с вещественными числами**

Для записи констант или при вводе-выводе может использоваться как привычное представление в виде десятичной дроби, например 123.456, так и "инженерная" запись числа, где мантисса записывается в виде вещественного числа с одной цифрой до точки и некоторым количеством цифр после точки, затем следует буква ''e'' (или ''E'') и экспонента. Число 123.456 в инженерной записи будет выглядеть как 1.23456e2, что означает, что 1.23456 нужно умножить на 10\*\*2. И мантисса и экспонента могут быть отрицательными и записываются в десятичной системе.

Такая запись чисел может применяться при создании вещественных констант, а также при вводе и выводе. Инженерная запись удобна для хранения очень больших или очень маленьких чисел, чтобы не считать количество нулей в начале или конце числа.

Если хочется вывести число не в инженерной записи, а с фиксированным количеством знаков после точки, то следует воспользоваться методом format, который имеет массу возможностей. Нам нужен только вывод фиксированного количества знаков, поэтому воспользуемся готовым рецептом для вывода 25 знаков после десятичной точки у числа 0.1:

| 1 | x = 0.1 |
| --- | --- |
| 2 | print('{0:.25f}'.format(x)) |

Вывод такой программы будет выглядеть как 0.1000000000000000055511151, что еще раз подтверждает мысль о том, что число 0.1 невозможно сохранить точно.

**Проблемы вещественных чисел**

Рассмотрим простой пример:

| 1 | if 0.1 + 0.2 == 0.3: |
| --- | --- |
| 2 | print('Yes') |
| 3 | else: |
| 4 | print('No') |

Если запустить эту программу, то можно легко убедиться в том, что 0.1 + 0.2 не равно 0.3. Хотя можно было надеятся, что несмотря на неточное представление, оно окажется одинаково неточным для всех чисел.

Поэтому при использовании вещественных чисел нужно следовать нескольким простым правилам:

1) Если можно обойтись без использования вещественных чисел - нужно это сделать. Вещественные числа проблемные, неточные и медленные.

2) Два вещественных числа равны между собой, если они отличаются не более чем на epsilon. Число X меньше числа Y, если X < Y - epsilon.

Код для сравнения двух чисел, заданных с точностью 6 знаков после точки, выглядит так:

| 1 | x = float(input()) |
| --- | --- |
| 2 | y = float(input()) |
| 3 | epsilon = 5 \* 10\*\*-7 |
| 4 | if abs(x - y) < 2 \* epsilon: |
| 5 | print('Equal') |
| 6 | else: |

В случае, если над числами совершались какие-то действия, то значения epsilon нужно вычислять как в приведенном в первом видео примере. В учебных задачах это можно сделать не внутри программы, а один раз руками для худшего случая и применять вычисленное значение как константу.

**Округление вещественных чисел**

При использовании целых и вещественных чисел в одном выражении вычисления производятся в вещественных числах. Тем не менее, иногда возникает необходимость преобразовать вещественное число в целое. Для этого можно использовать несколько видов функций округления:

int - округляет в сторону нуля (отбрасывет дробную часть)

round - округляет до ближайшего целого, если ближайших целых несколько (дробная часть равно 0.5), то к чётному

floor - округляет в меньшую сторону

ceil - округляет в большую сторону

Примеры для различных чисел:

| **Функция** | **2.5** | **3.5** | **-2.5** |
| --- | --- | --- | --- |
| int | 2 | 3 | -2 |
| round | 2 | 4 | -2 |
| floor | 2 | 3 | -3 |
| ceil | 3 | 4 | -2 |

Функции floor и ceil находятся в библиотеке math. Есть два способа получить воспользоваться ими в своей программе.

В первом способе импортируется библиотека math, тогда перед каждым вызовом функции оттуда нужно писать слово ''math.'', а затем имя функции:

| 1 | import math |
| --- | --- |
| 2 |  |
| 3 | print(math.floor(-2.5)) |
| 4 | print(math.ceil(-2.5)) |

Во втором способе из библиотеки импортируются некоторые функции и доступ к ним можно получить без написания ''math.'':

| 1 | from math import floor, ceil |
| --- | --- |
| 2 |  |
| 3 | print(floor(-2.5)) |
| 4 | print(ceil(-2.5)) |

Второй способ предпочтительно применять в случае, если какие-то функции используются часто и нет конфликта имен (функций с одинаковыми именами в нескольких подключенных библиотеках).

В библиотеке math также есть функция округления trunc, которая работает аналогично int.

**Срезы строк**

Нам известны способы считывать, выводить и задавать константные строки, а также склеивать строки между собой и умножать строку на число.

Чтобы определить длину строки s можно воспользоваться функцией len(s) - она возвращает целое число, равное длине строки.

Срез - это способ извлечь из строки отдельные символы или подстроки. При применении среза конструируется новая строка, строка, к которой был применён срез, остается без изменений.

Простейший вид среза - это обращение к конкретному символу строки по номеру. Чтобы получить i-ый символ строки нужно написать s[i]. В результате этого будет сконструирована строка, содержащая только один символ --- тот, который стоял на месте i. Нумерация символов идет с нуля, при попытке обратиться к символу с номером больше либо равном длине строки возникает ошибка.

В языке Питон присутствует и нумерация символов строки отрицательными числами. Последний символ строки имеет номер -1, предпоследний -2 и так далее. При попытке обратиться к символу с номером, меньшим чем -len(s) возникает ошибка.

Нумерация символов в строке ''String'' представлена в таблице:

| S | t | r | i | n | g |
| --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 |
| -6 | -5 | -4 | -3 | -2 | -1 |

Получить доступ, например, к символу n, можно двумя способами s[4] и s[-2].

Также существуют срезы с двумя параметрами: в результате применения среза s[a:b] будет сконструирована подстрока начиная с символа на позиции a и заканчивая символом на позиции b-1 (правая граница не включается). Каждый из индексов может быть как положительным, так и отрицательным.

Например, при s = "String", s[1:5] будет равно "trin", это можно было бы записать и как s[1:-1]. Если в качестве второго числа в срезе взять число, больше либо равное длине строки, то ошибки не возникнет и будут взяты все символы до конца строки.

В случае, если нужно взять все символы строки начиная с позиции a и до конца, то второй параметр можно опускать. Например, s[2:] будет равно "ring".

Если опустить первый параметр в срезе, то будет взята подстрока с начала, например s[:2] будет равно "St". Если же написать S[:] то будет взята вся строка от начала до конца.

Если первый параметр находится правее второго, то будет сгененрирована пустая строка.

Также существует срез с тремя параметрами, где третий параметр задает шаг, с которым нужно брать символы. Например, можно взять все символы с начала до конца с шагом 2, это будет выглядеть как s[::2], в результате чего получится строка "Srn". Естествено, первый и второй параметры можно не опускать. Если третий параметр не указан, т.е. в квадратных скобках записано только одно двоеточие, то шаг считается равным 1.

Шаг в срезе может быть и отрицательным, в таком случае первый параметр должен находится правеее второго. Например, s[5:1:-2] даст строку "gi - 5-ый и 3-ий символы, а символ с номером 1 уже не входит. Развернутую строку можно получить срезом s[::-1] -все символы от "начала" до "конца" в обратном порядке. Если третий параметр отрицательный, то началом среза считается последний символ, а концом - позиция перед нулевым символом.

**Метод find**

Методы - это функции, применяемые к объектам. Метод вызывается с помощью записи ИмяОбъекта.НазваниеМетода(Параметры). Методы очень похожи на функции, но позволяют лучшим образом организовывать хранение и обработку данных. Например, вы написали свою структуру данных и хотели бы, чтобы функция len возвращала длину вашей структуры. Чтобы это заработало, вам придется лезть в исходный код интерпретатора Питона и вносить изменения в функцию len. Если бы len было методом, то вы могли бы описать этот метод при создании структуры и никаких изменений в коде интерпретатора или стандартной библиотеки не потребовалось бы. Поэтому методы предпочтительнее для сложных структур, например, таких как строки.

У строк есть множество различных методов. В этом разделе мы рассмотрим методы поиска подстроки в строке. Метод find возвращает индекс первого вхождения подстроки в строку, а если она не нашлась -1. Например, 'String'.find('ing') вернет 3 - индекс, с которого начинается вхождение подстроки ing.

Существуют модификации этих методов с двумя параметрами. s.find(substring, from) будет осуществлять поиск в подстроке s[from:]. Например, 'String'.find('ing', 1) вернет 3 (нумерация символов остается как в исходной строке). По аналогии со срезами параметры могут быть и отрицательными.

Также есть модификации с тремя параметрами: они ищут подстроку в срезе s[a:b].

Часто возникает задача найти и вывести все вхождения подстроки в строку, включая накладывающиеся. Например, для строки 'ABABA' и подстроки 'ABA' ответ должен быть 0, 2. Ее решение выглядит так:

| 1 | string = input() |
| --- | --- |
| 2 | substring = input() |
| 3 | pos = string.find(substring) |
| 4 | while pos != -1: |
| 5 | print(pos) |
| 6 | pos = string.find(substring, pos + 1) |

**Методы rfind, replace и count**

Метод rfind работает аналогично find, но ищет самое правое вхождение.

Метод replace(old, new) позволяет заменить все вхождения подстроки old на подстроку new. При этом конструируется новая строка, где были произведены замены. Нужно обратить внимание, что метод replace заменяет вхождения подстрок без учета предыдущих совершенных замен. Если применить следующую операцию 'AAAAAA'.replace('AA', 'A'), то в результате получится строка 'AAA', а не 'A', как можно было бы ожидать.

Фактически, можно считать, что метод replace находит очередное вхождение подстроки old, осуществляет замену и продолжает поиск с позиции после всех замененных символов (без наложения и поиска в свежезамененной части).

Существует модификация replace(old, new, count), которая осуществляет не более count замен самых левых вхождений подстроки old.

Также для строк существует метод count, который позволяет подсчитать количество вхождений подстроки. По аналогии с методом find определены методы count с двумя и тремя параметрами.

Последнее изменение: Thursday, 6 September 2018, 05:13