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 значащими 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.

53 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 сравнения вещественных чисел.

Для записи констант или при вводе-выводе может использоваться как привычное представление в виде десятично 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:

```
x = 0.1
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:

```
if 0.1 + 0.2 == 0.3:
    print('Yes')
else:
    print('No')
```

Если запустить эту программу, то можно легко убедиться в том, что 0.1 + 0.2 не равно 0.3. Хотя можно было надеяться, 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) Две 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:

```
x = float(input())
y = float(input())
epsilon = 5 * 10**-7
if abs(x - y) < 2 * epsilon:
    print('Equal')
else:
    print('Not equal')
```

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:

```
import math
print(math.floor(-2.5))
print(math.ceil(-2.5))
```

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

```
from math import floor, ceil
print(floor(-2.5))
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.