

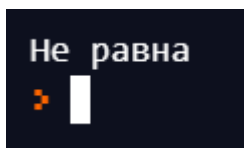
Урок 4. Особенности работы с вещественными числами

Итак, мы уже разобрались с тем, как записываются очень большие и очень маленькие числа в Python. Однако мы до сих пор не знаем, как именно они представляются внутри самого компьютера.

Пример 1

```
main.py
1  if ((1.1 + 2.2) == 3.3):
2      print('Равна')
3  else:
4      print('Не равна')
5
```

Давайте для начала рассмотрим вот такое выражение: $(1.1 + 2.2) == 3.3$. Для нас очевидно, что сумма 1.1 и 2.2 равна 3.3. Давайте проверим.



```
Не равна
>
```

Похоже, Python с этим не согласен. Странно, не так ли? Давайте попробуем вывести на экран просто сумму этих двух чисел.



```
main.py
1  print(1.1 + 2.2)
2
3
```

Console

```
3.3000000000000003
>
```

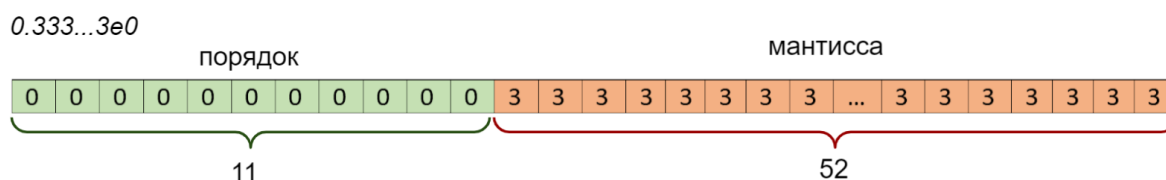
И тут ответ нас удивит: после тройки у нас идёт куча нулей и ещё одна тройка в конце. И получается, что это число действительно не будет равняться 3.3. Но как же так? Чтобы понять, что тут происходит, нам нужно разобраться, как числа хранятся в памяти компьютера.

Погрешности

$$\frac{1}{3} \quad 0.3 \quad 0.33 \quad 0.333 \quad 0.33333333\dots$$

Рассмотрим дробь $1/3$. В виде десятичной дроби мы можем приблизительно представить её как 0.3. А если быть точнее, то можно так: 0.33. А если быть ещё точнее, то так: 0.333. Ну и так далее, можно продолжать до бесконечности. Вот только независимо от того, как много цифр мы запишем, результат никогда не будет точно $1/3$, но будет всё более лучшим к нему приближением.

Дело в том, что это число представляется в компьютере примерно вот так:



Мы уже говорили про порядок и мантиссу, когда обсуждали экспоненциальное представление числа в программе. Так вот, с помощью тех же порядка и мантиссы числа представляются и внутри компьютера. И, как мы уже знаем, в компьютере есть ограничение на хранение чисел. Как мы видим по рисунку, это означает, что, к сожалению, **большинство десятичных дробей не могут быть точно представлены в двоичной записи.**

$$\frac{1}{2} = 0.5 \quad \frac{2}{3} \approx 0.666666666666666666$$

Одни дробные значения могут быть представлены точно (такие как 0.5), а другие — только приблизительно. Следствием этого является то, что в основном десятичные дробные числа мы вводим только приближенными к двоичным, которые и сохраняются в компьютере.

В связи с этим (и некоторыми другими фактами) **числа типа float не могут надёжно сравниваться на равенство значений, так как имеют ограниченную точность.**

$$1.1 + 2.2 \neq 3.3$$

Не рекомендуется

Или, к примеру, если мы в результате вычислений получили дробные числа вроде 0.12345645340568035860493605 и 0.39825093485092348598209345, то при их сложении или вычитании мы также получим не самое точное число. Проблема потери точности это не проблема самого языка Python, а особенность компьютерного представления чисел.

Пример 2

main.py

```
1  a = 1.0
2
3  while True:
4      a+= 1e-324
5      print(a)
6
```

Теперь ещё давайте посмотрим вот на такую программу. Предположим, к нашему числу а нужно очень много раз прибавлять супермаленькое число. Примерно то самое число, которое мы получили при постоянном делении на 2, то есть 10 в минус 324 степени. Подобная задача, кстати, вполне себе имеет место, но об этом мы будем говорить чуть позже. Давайте попробуем запустить эту программу.

```
1.0
1.0
1.0
1.0
1.0
1.0
```

и так до бесконечности...

Как-то странно, переменная «а» у нас не меняется и всегда равна единице. Давайте попробуем немного увеличить наше маленькое число и сделаем не -324, а -20.

```
main.py
1  a = 1.0
2
3  while True:
4      a+= 1e-20
5      print(a)
6
```

Console

```
1.0
1.0
1.0
1.0
1.0
```

Всё ещё плохо. Может быть, попробовать взять ещё меньше? Скажем, -10.

```
main.py ×
1  a = 1.0
2
3  while True:
4      a+= 1e-10
5      print(a)
6
```

Console

```
1.0000037258003083
1.0000037259003083
1.0000037260003083
1.0000037261003083
1.0000037262003083
1.0000037263003083
```

Здесь уже всё работает, как надо. Интересно, а если взять -15?

```
main.py
1  a = 1.0
2
3  while True:
4      a+= 1e-15
5      print(a)
6
```

Console

```
1.0000000000116573
1.0000000000116585
1.0000000000116596
1.0000000000116607
1.0000000000116618
1.0000000000116629
```

Снова всё в порядке. Значит, где-то между -16 и -20 что-то ломается. Вариантов тут немного, поэтому давайте сразу попробуем -16.

```
main.py
1  a = 1.0
2
3  while True:
4      a+= 1e-16
5      print(a)
6
```

Console

```
1.0
1.0
1.0
1.0
1.0
```

Ага, и здесь наша программа уже сломалась! Давайте разберёмся подробнее, как так получается и в чём здесь дело.

Представление чисел (часть 1)

1	0	0	0	2	5	6
---	---	---	---	---	---	---

Предположим, что мы сделали свой компьютер и для хранения чисел в нём выделили семь ячеек в памяти. Возьмём любое число, которое в эти ячейки поместится, например 1000256. И теперь мы захотели прибавить к нему какое-то другое число, например 10000128. Если считать как обычно, то по идее получится 11000384.

	1	0	0	0	2	5	6
+							
1	0	0	0	0	1	2	8
1	1	0	0	0	3	8	4

Но тут мы видим, что в получившемся числе разрядов больше, чем ячеек в нашей памяти. И тут возникает вопрос: а что в таком случае происходит? Ячеек у нас семь, а разрядов восемь, значит, лишний нужно отбросить. Лишний у нас будет либо слева, либо справа.

1	1	0	0	0	3	8	4
---	---	---	---	---	---	---	---

 либо

1	1	0	0	0	3	8	4
---	---	---	---	---	---	---	---

Какой же тогда убирается? Так как в числе мы всегда идём слева направо, то есть от старшего разряда к младшему, то отбросится именно правая часть. И в ответе получится 1100038, а четвёрка просто занулится в компьютере.

1	1	0	0	0	3	8	0
---	---	---	---	---	---	---	---

Точно так же всё работает и в обычном компьютере, просто ячеек памяти там не семь, а гораздо больше.

Представление чисел (часть 2)

Кстати, здесь же мы подходим к ещё одной интересной штуке. Снова возьмём наше число 1000256 и семь ячеек памяти. Какое самое маленькое число мы могли бы к нему прибавить? Конечно же, это единица. И записана она в памяти будет как шесть нулей и 1.

$$\begin{array}{r}
 \begin{array}{|c|c|c|c|c|c|c|} \hline 1 & 0 & 0 & 0 & 2 & 5 & 6 \\ \hline \end{array} \\
 + \\
 \begin{array}{ccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array} \\
 \\
 \begin{array}{cccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array} \\
 = \\
 \begin{array}{|c|c|c|c|c|c|c|} \hline 1 & 0 & 0 & 0 & 2 & 5 & 6 \\ \hline \end{array}
 \end{array}$$

А что, если эта единица будет записана в виде восьми разрядов, как в третьей строчке? Что тогда? В таком случае получится, что наш компьютер просто не увидит эту единицу и воспримет её как ноль. А значит, и к нашему числу прибавится ноль.

$$\begin{array}{r}
 \begin{array}{|c|c|c|c|c|c|c|} \hline 1 & 0 & 0 & 0 & 2 & 5 & 6 \\ \hline \end{array} \\
 + \\
 \begin{array}{ccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array} \leftarrow \text{Машинный эпсилон } (\epsilon) - \text{предельно маленькое} \\
 \text{различимое вещественное число}
 \end{array}$$

В таком случае вторая строчка (число 0000001) называется **машинным эпсилоном**. Это минимально различимое для программы число. Для компьютера всё, что меньше эпсилона, эквивалентно нулю. Именно такое число мы и нашли здесь, а также нашли в нашей программе. Там оно было равно $1e-16$.

$$x < \epsilon? \Rightarrow x == 0$$

$$1e-16$$

$$\text{Если } B < \epsilon, \text{ то } A + B == A$$

$$\text{Если } A - B < \epsilon, \text{ то } A == B$$

Если добавить число, меньше эпсилона, к любому другому, то это будет всё равно что добавить ноль. Процессор не увидит разницы. Именно поэтому наша переменная A никак не хотела меняться. Точно так же можно сказать, что если два вещественных числа A и B отличаются меньше чем на машинный эпсилон, компьютер их тоже не различит. **Для типа float этот эпсилон находится в районе шестнадцатого знака после запятой** — примерно с такой точностью мы и работаем, когда используем этот тип данных в Python.

Представление чисел (часть 3)

Вы можете здесь справедливо заметить: «Вообще-то тема нашего урока связана с вещественными числами. А мы посмотрели на примеры целых. При чём тут они?». На

самом деле, мы уже говорили о нечто подобном, когда работали с экспоненциальной формой числа. Давайте для примера возьмём вот такое число с плавающей точкой.

0	.	0	0	0	0	0	0	1	1	0	0	3	8	4
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Мы видим, что в дробной части сначала идёт шесть нулей, а потом какое-то число. Так вот, по сути, компьютер так и хранит вещественные числа в памяти: он в целочисленном типе записывает количество нулей после точки и так же целочисленно записывает само число. Вот это количество нулей как раз и будет наш порядок, а число — это наша мантисса!

Нулей: 6 ← Порядок

Число: 1100384 ← Мантисса

0.1100384e-6

И здесь мы как раз сталкиваемся с тем, что мы обсуждали пару минут назад. Под эти порядок и число ведь нужно выделять память! А память у нас не резиновая, и у неё есть предел.

Нулей:

0	0	6
---	---	---

Число:

1	1	0	0	3	8	4
---	---	---	---	---	---	---

Таким образом, наше число, то есть мантисса, может просто не влезть, и тогда мы потеряем в точности. По этой же причине при сложении или вычитании таких чисел мы также получим не самое точное число, о чём мы уже говорили. И также, смещая точку в разные места, у нас будет меняться наше минимально различимое число, то есть эпсилон.

Продолжение примера 1

```
main.py
1  if ((1.1 + 2.2) == 3.3):
2      print('Равна')
3  else:
4      print('Не равна')
5
```

Теперь давайте снова вернёмся к нашему сравнению. Только слегка его перепишем для удобства понимания: вынесем наши части, которые сравниваем, в разные переменные:

main.py

```
1  a = 1.1
2  b = 2.2
3  c = a + b
4  d = 3.3
5  if (c == d):
6      print('Равна')
7  else:
8      print('Не равна')
9
```

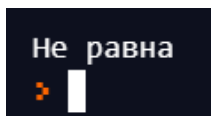
В связи с тем, что нам выдаётся тот же ответ «Не равна», то возникает вопрос: «Что нам теперь делать? Как тогда сравнивать нужные нам числа?» Давайте подумаем. У нас есть равенство. По идее, если перенести d влево, то останется ноль. Давайте попробуем:

```
5  if (c - d == 0):
```

Вообще, вместо этих чисел в наших переменных могут быть какие угодно, значит, нам следует это учесть. Ведь сравнивать нам нужно будет только положительные числа. А значит, можно просто взять модуль от левого выражения:

```
5  if abs(c - d) == 0:
```

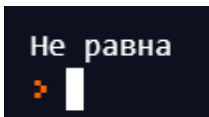
Тест:

A terminal window with a dark background. It displays the text "Не равна" in white. Below the text is a small orange prompt character and a white cursor line.

Как видим, ответ всё тот же, просто теперь всё наше выражение не равно нулю. Вопрос всё тот же: «Как нам сравнивать?» И здесь нам пригодится тот самый машинный эпсилон, про который мы говорили в прошлом уроке. Мы помним, что если два вещественных числа отличаются меньше чем на машинный эпсилон, компьютер их тоже не различит. Давайте попробуем сравнить выражение с эпсилоном:

```
5  if abs(c - d) < 1e-16:
```

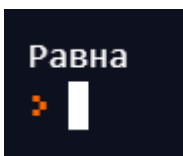

Тест:



Кажется, ответ опять тот же. И на самом деле, так и должно быть, ведь эпсилон — это предельное маленькое число, а наше выражение уж точно не может быть меньше предельного числа. В таком случае давайте попробуем немного изменить степень — сделаем не -16, а -15:

```
5 if abs(c - d) < 1e-15:
```

Тест:



Кажется, теперь всё заработало! Таким образом, чтобы сравнить нужные нам числа, нужно взять модуль их разности и посмотреть, будет ли эта разность меньше какого-то очень маленького числа. Кстати, то, что мы сейчас сделали, называется **заданием точности**, и в следующем уроке мы как раз поговорим, зачем это может быть нужно и как это используется.

Итоги урока

На этом наш урок завершается. Теперь мы понимаем фундаментальные сложности, с которыми сопряжена работа с вещественными числами в программировании. Давайте вспомним ключевые правила обращения с ними:

- Мы не можем получить сколь угодно малое и сколь угодно большое число. В Python мы можем достичь в лучшем случае вот таких чисел.
- Мы не можем получить число сколько угодно большой точности. В Python, как и в других языках, есть ограничение на представление числа.
- Очень нежелательно проверять два вещественных числа на равенство. Вместо этого надо пользоваться математическим лайфхаком.
- Если порядки чисел слишком разные, то мы не можем их складывать: нам не хватит точности и большое число просто поглотит маленькое.
- Следствие из последней мысли: даже если порядки не слишком отличаются, то часть меньшего числа всё равно потеряется из-за той же проблемы с точностью.