1.2 Вещественные числа и математические вычисления

1.2.1 Дополнительные возможности библиотеки math

В предыдущей теме мы уже вскользь знакомились с библиотекой math и использовали её функции, такие как pow() и abs(). В этой теме мы рассмотрим другие функции этой библиотеки, а также возможность использования математической константы.

```
import math
print(math.sqrt(4)) # 2.0
```

В примере выше мы импортировали модуль math и сделали что-то новое. Мы вызвали одну из функций модуля – sqrt (т.е. square root – квадратный корень). Для вызова метода импортированного модуля, нам нужно использовать следующий синтаксис: module_name.method_name(аргумент). В данном примере мы нашли квадратный корень от 4. В модуле math есть много других функций, которыми мы можем воспользоваться, такие как нахождение косинуса, факториал, логарифмы и другие. Вы можете призывать эти функции таким же образом, как и с функцией sqrt. Единственное, в чем вам нужно удостовериться – принимают ли они большее количество аргументов или нет. Теперь посмотрим на другой способ импорта.

Некоторые люди не очень любят обозначать все, что они пишут названиями модулей. Впрочем, в python есть решение и для этого. Вы можете импортировать из модуля только те функции, которые вам нужны. Представим, что нам нужно импортировать только функцию sqrt:

```
from math import sqrt
print( sqrt(16) ) # 4.0
```

Это работает именно так, как читается: функция sqrt импортируется из модуля math. Попробую объяснить это иначе. Мы использовали ключевое слово from для импорта функции sqrt из модуля math. Мы также можем использовать этот метод для импорта нескольких функций из модуля math:

```
from math import pi, sqrt
```

В данном примере мы импортировали как sqrt так и pi. Перед тем как обратиться к pi, стоит обратить внимание на то, что это не функция, которую вы вызываете, а величина.

В python предусмотрена возможность импорта всех функций, переменных и модулей за раз. Это может быть плохой идеей, так как такой импорт может засорить ваше пространство имен. Пространство имен – это место, в котором находятся все ваши переменные, пока работает программа. К примеру, у вас есть своя переменная под названием sqrt:

```
from math import sqrt
sqrt = 5
```

Только что вы сделали функцию sqrt переменной, значение которой 5. Это называется затенением (shadowing). Это особенно хитрый способ упрощения своей жизни, когда вы выполняете импорт всего из модуля. Давайте взглянем:

```
from math import *

sqrt = 5
sqrt(16)
```

Будет ошибка!

```
Traceback (most recent call last):
File “<string>”, line 1, in <fragment>
TypeError: ‘int’ object is not callable
```

Для импорта всего, кроме определенных частей модуля, мы просто используем знак * , что означает, что нам нужно импортировать все. В случае если мы не знаем, что находится в модуле math, мы даже не поймем, что уничтожили одну из импортированных функций. Когда мы пытаемся вызвать функцию sqrt после присвоения её целому числу, мы обнаружим, что она больше не работает.

По этой причине, в большинстве случаев рекомендуется использовать один из указанных в данном разделе способов для импорта.

Если говорить о тригонометрических функциях библиотеки math, то они выглядят следующим образом:

- `math.cos()`
- `math.sin()`
- `math.tan()`
- `math.acos()`
- `math.asin()`
- `math.atan()`

Обратите внимание! Все значения, которые возвращают эти функции, измеряются в радианах.

1.2.2 Округление вещественных чисел

Числа с плавающей точкой — быстрый и эффективный способ хранения чисел и работы с ними. Но он связан с рядом трудностей для начинающих и опытных программистов! Вот классический пример:

```
>>> 0.1 + 0.2 == 0.3
False
```

Впервые увидев такое, можно растеряться. Такое поведение корректно! Поговорим о том, почему ошибки при операциях над числами с плавающей

точкой так распространены, почему они возникают и как с ними справиться в Python. Проблема касается и сравнения:

```
>>> 0.1 + 0.2 <= 0.3
False
```

Что происходит? Когда вы вводите в интерпретатор Python число 0.1, оно сохраняется в памяти как число с плавающей точкой и происходит преобразование. 0.1 — это десятичное число с основанием 10, но числа с плавающей точкой хранятся в двоичной записи. То есть основание 0.1 преобразуется из 10 в 2.

Получающееся двоичное число может недостаточно точно представлять исходное число с основанием 10. 0.1 — один из примеров. Двоичным представлением будет 0.0(0011). То есть 0.1 — это бесконечно повторяющееся десятичное число, записанное с основанием 2. То же происходит, когда в виде десятичного числа с основанием 10 записывается дробь ⅓. Получается бесконечно повторяющееся десятичное число 0.3(3).

Память компьютера конечна, поэтому бесконечно повторяющееся представление двоичной дроби 0.1 округляется до конечной дроби. Её значение зависит от архитектуры компьютера (32- или 64-разрядная).

При сложении 0.1 и 0.2 получается число чуть больше 0.3:

```
>>> 0.1 + 0.2
0.30000000000000004
```

А поскольку 0.1 + 0.2 чуть больше, чем 0.3, и 0.3 представлено числом, которое чуть меньше 0.3, выражение 0.1 + 0.2 == 0.3 оказывается False.

Об ошибке представления чисел с плавающей точкой должен знать каждый программист на любом языке — и уметь с ней справляться. Она характерна не только для Python.

Как же справляться с ошибками представления чисел с плавающей точкой при сравнении таких чисел в Python? Хитрость заключается в том, чтобы избегать проверки на равенство. Вместо ==, >= или <= всегда используйте с числами с плавающей точкой функцию math.isclose():

```
>>> import math
>>> math.isclose(0.1 + 0.2, 0.3)
True
```

В math.isclose() проверяется, достаточно ли близок первый аргумент ко второму. То есть проверяется расстояние между двумя аргументами. Оно равно абсолютной величине разницы обоих значений:

```
>>> a = 0.1 + 0.2
>>> b = 0.3
>>> abs(a - b)
5.551115123125783e-17
```

Если abs(a - b) меньше некоего процента от большего значения a или b, то a считается достаточно близким к b, чтобы считаться «равным» b. Этот процент называется относительной погрешностью и указывается именованным аргументом rel_tol функции math.isclose(), который по умолчанию равен 1e-9.

То есть если abs(a - b) меньше 0.00000001 * max(abs(a), abs(b)), то a и b считаются «близкими» друг к другу. Это гарантирует, что a и b будут приблизительно с девятью знаками после запятой.

Другим, пусть и грубым, способом сравнения вещественных чисел является округление этих самых чисел до нужного количества знаков после запятой. Делается это при помощи функции format(). Синтаксис этой функции следующий:

```
format(число_или_выражение, '.nf'),
```

где вместо n пишется необходимое количество знаков после запятой:

```
>>> print(format(2 / 3, '.3f'))
0.667
```

Пример решения вышеупомянутой задачи:

```
>>> a = format(0.1 + 0.2, '.1f')
>>> b = format(0.3, '.1f')
>>> if a == b:
>>>     print("OK")
>>> else:
>>>     print("Not OK")
OK
```