Theory: Float special values

17 minutes 0 / 5 problems solved

When thinking about numbers, we tend to imagine something practical and specific, rather than abstract. The first thought might be to count things: how old are we, how many animal species are there on our planet and how many stars in the sky. It comes naturally to our minds, that's why we put the label *natural* on those numbers. Yet, there's more to the picture. Counting is not the only thing we are good at. For example, we can measure physical quantities to explore the world around us. These real-life challenges are better described by *real* numbers. As you know, *floats* represent real numbers in Python, but they have more to offer. Today we will discuss some abstract concepts and find out in what respect the Python floats have gone beyond the real number line.

§1. What is infinity?

The concept of infinity might be somewhat difficult to perceive. Let's try to visualize it! We can take our galaxy for starters. Is the Milky Way big enough? Until about 100 years ago, you might have thought so, but the answer is different now (here is a timeline to consult with). So what comes next: two galaxies, a galactic cluster, the universe, the multiverse? Let's stop right there and highlight the pattern of such reasoning.

When we call something infinite what we basically mean is that it is always possible to add one more. No matter how large is the number we come up with, the infinite value will be greater than it. Say, we've picked a really large number ***n***, well, there will always be ***n+1*** and so on.

§2. Infinity constants

In Python, infinity is a constant greater than any real number. This special value is denoted as `inf`. Where can you encounter it? For example, you intend to convert `10^3000` to float:

```
print(float(10 ** 3000))  # OverflowError
print(float("1e3000"))    # inf
```

This number is too large to be represented, but you can still convert it to float if you pass it as a floating-point literal. The second line doesn't generate an overflow and returns `inf` instead, which is pleasant because you can work with it and perform further arithmetic operations.

If you want to experiment with an infinite value, there is a better way to obtain one. Just use floating-point literals again: either `"infinity"` or `"inf"`. The case doesn't matter, what is more, your string may contain leading and trailing whitespace characters:

```
inf_var = float("INFINITY    ")  # inf

inf_war = float("INFINITY WAR")  # ValueError: could not convert string to float
```

Infinity has a counterpart `-inf`. The negative infinity constant is less than any real number. You can get it just as easily:

```
print(float("-inf"))  # -inf
```

The constants `inf` and `-inf` support basic arithmetic operations, as all floating-point numbers do:

Current topic:

Float special values

Topic depends on:

- ✓ Identity testing
- ✓ Exceptions Stage 1 3★
- ✗ Math functions

Table of contents:

```
inf = float("inf")

# addition
print(inf + inf)   # inf
print(inf + 1)     # inf

# multiplication
print(-inf * -66)  # inf
print(-inf * inf)  # -inf



# subtraction

print(-inf - inf)  # -inf

print(inf - 9999)  # inf
```

Let us dwell on a few details. First, you can't have two infinities or double infinity. It's an ever-increasing value in case of `inf`, and sort of the ever-decreasing one when it comes to `-inf`. Both constants are part of the floating-point arithmetic system, so its rules will generally apply. In case of addition, multiplication or subtraction, you are likely to get infinity (the sign depends).

```
# division
print(-inf / 100)  # -inf
print(inf / inf)   # nan
print(inf / 0.0)   # ZeroDivisionError

# integer division
print(1 // inf)    # 0.0
print(inf // 1)    # nan
print(5.0 // 0)    # ZeroDivisionError
```

The division examples may also prompt some questions. In Python, division by zero always raises `ZeroDivisionError`, however, in other programming languages, you can get away with this if at least one of the operands is a real number. Another point to clarify is the mysterious `nan`, to which we devote the next section.

§3. Meet NaN

Floating-point numbers include another special value called `NaN`, which stands for *Not a Number*. `NaN` is not equal to any floating-point number (including *itself* and ±*inf*). The idea behind `NaN` is that forbidden arithmetic operations still need a value to return. Here is an example:

```
pos_inf = float("+inf")
neg_inf = float("-inf")

print(pos_inf + neg_inf)  # nan
```

As you are well aware, multiplication by zero results in zero. The `NaN` value is even more contagious: most arithmetic operations on it return `NaN`. However, a counter-example is `NaN ** 0`, it produces *1.0*.

> The systematic use of *NaN* and *infinity constants* was introduced by the **IEEE 754** standard in 1985. IEEE 754 specified how numeric values should be represented in computer hardware and addressed many issues found in diverse floating-point implementations.

§4. NaN vs None

Although, the constants `NaN` and `None` sound similar, there is a difference between them. `None` is a singleton object that denotes no value at all. `None` evaluates to `False` in boolean expressions, while `NaN` doesn't. As opposed to `None`, `NaN` is different in every given case, it never equals to itself: you should consider `NaN != NaN` true.

The real-world application of `NaN` is related to the lack of numerical data. Missing values crop up in datasets due to numerous reasons. When the data is insufficient, `NaN` is commonly used to substitute missing floating-point values. Try not to mistake it for `None`.

§5. math.isnan()

Since the equality test `NaN == NaN` doesn't work as intended, you might wonder how to test for `NaN` values. There is a solution in the standard library. The function `isnan(x)` from the *math* module determines whether *x* is a floating-point *"not a number"* value.

```
import math


inf = math.inf   # an alternative way
nan = math.nan   # to get the constants

math.isnan(nan)  # True
math.isnan(inf)  # False
math.isnan(0.0)  # False
```

The function returns `True` only when its argument is `NaN`.

§6. math.isinf()

The *math* module also provides a function `isinf()` to check for an *infinity* value (either positive or negative).

```
math.isinf(inf)   # True
math.isinf(-inf)  # True
math.isinf(3.14)  # False
```

As seen from the example, `math.isinf()` works for both `inf` and `-inf` and returns a boolean value.

§7. math.isfinite()

The function `math.isfinite(x)` returns `True` if *x* is neither an *infinity* value nor a *NaN*, and `False` otherwise.

```
math.isfinite(7.54)  # True
math.isfinite(inf)   # False
math.isfinite(nan)   # False
```

`math.isfinite()` helps find the floating-point numbers distinct from the special values.

Similar functions can be found at other resources. For example, the library for scientific computing NumPy provides the following functions: *isinf*, *isposinf*, *isneginf*, *isnan* and *isfinite*. We recommend that you use some of the mentioned functions, for they are more likely to return the up-to-standard result.

§8. Summary

Let's go over the main points:

- Python floats include the special values `inf`, `-inf` and `nan`.
- The `inf` constant is greater than any real number. To create it, use `float("inf")` or `math.inf`.
- The `-inf` constant is less than any floating-point number. You can obtain it via `float("-inf")` or `-math.inf`.
- The `nan` constant stands for *"not a number"* and is not equal to any float. Use `float("nan")` or `math.nan` to get it.
- The *math* module has functions designed for special value tests: `math.isinf()`, `math.isnan()`, `math.isfinite()`.

Report a typo

92 users liked this theory. **0** didn't like it. **What about you?**

Start practicing

Comments (2) Hints (0) Useful links (0) Show discussion