

Programmers Stack Exchange is a question and answer site for professional programmers interested in conceptual questions about software development. It's 100% free, no registration required.

Take the 2-minute tour

×

How to handle divide by zero in a language that doesn't support exceptions?

I'm in the middle of developing a new programming language to solve some business requirements, and this language is targeted at novice users. So there is no support for exception handling in the language, and I wouldn't expect them to use it even if I added it.

I've reached the point where I have to implement the divide operator, and I'm wondering how to best handle a divide by zero error?

I seem to have only three possible ways to handle this case.

1. Ignore the error and produce `0` as the result. Logging a warning if possible.
2. Add `NaN` as a possible value for numbers, but that raises questions about how to handle `NaN` values in other areas of the language.
3. Terminate the execution of the program and report to the user a severe error occurred.

Option #1 seems the only reasonable solution. Option #3 is not practical as this language will be used to run logic as a nightly cron.

What are my alternatives to handling a divide by zero error, and what are the risks with going with option #1.

programming-languages

error-handling

asked Aug 18 '13 at 11:45

 **Mathew Foscarini**

8,949 2 28 51

11 if you did add exception support and the user didn't catch it then you'd have option #3 – **ratchet freak** Aug 18 '13 at 11:54

81 I'm curious, what kind of stupid requirement would require you to create a whole new programming language? In my experience, every language ever created sucks (in design or in execution, often in both) and it took unreasonably much effort to even get *that* much. There are a few exceptions to the first, but not to the second, and as they're easily <0.01% of the cases, they're probably measurement errors ;-)

– **delnan** Aug 18 '13 at 12:04

15 @delnan most new languages are created to allow business rules to be separated from how they are implemented. The user does not need to know how `reject "Foo"` was implemented, but simply that it rejects a document if it contains the keyword `Foo`. I try to make the language as easy to read using terms the user is familiar with. Giving a user their own programming language empowers them to add business rules without depending upon technical staff.

– **Mathew Foscarini** Aug 18 '13 at 12:15

19 @Mathew Foscarini. Never, ever, ignore the error and silently return 0. When doing a division, 0 may be a perfectly legal value (for some reason, there is such a thing in Power Basic, and it's really a pain). If you divie floating point numbers, Nan or Inf would be nice (have a look at **IEEE 754** to understand why). If you divide integers, you may stop the program, divide by 0 should never be allowed (well, unless you want to implement a true exception system).

– **Jean-Claude Arbaut** Aug 19 '13 at 6:07

15 I amused and fascinated by a business domain complex enough to justify a proprietary, Turing-complete programming language but lax enough to tolerate drastically inaccurate results.

– **mehaase** Jul 12 at 20:06

16 Answers

I would strongly advise against #1, because just ignoring errors is a dangerous anti-pattern. It can lead to hard-to-analyze bugs. Setting the result of a division by zero to 0 makes no sense whatsoever, and continuing program execution with a nonsensical value is going to cause trouble. *Especially* when the program is running unattended. When the program interpreter notices that there is an error in the program (and a division-by-zero is almost always a design error), aborting it and keeping everything as-is is usually preferred over filling your database with garbage.


Also, you will unlikely be successful with thoroughly following this pattern through. Sooner or later you will run into error situations which just can't be ignored (like running out of memory or a stack overflow) and you will have to implement a way to terminate the program anyway.




Option #2 (using NaN) would be a bit of work, but not as much as you might think. How to handle NaN in different calculations is well-documented in the IEEE 754 standard, so you can likely just do

what the language your interpreter is written in does.


By the way: Creating a programming language usable by non-programmers is something we've been trying to do since 1964 (Dartmouth BASIC). So far, we've been unsuccessful. But good luck anyway.




edited Jul 11 at 19:31

 **SamtheBrand**

351  2  10  22

answered Aug 18 '13 at 12:48

 **Philipp**

6,574  1  11  27

13 +1 thanks. You convinced me to throw an error, and now that I read your answer I don't understand why I was hesitating. `PHP` has been a bad influence on me. – **Mathew Foscarini** Aug 18 '13 at 14:17

21 Yes, it has. When I read your question, I immediately thought that it was a very PHP-esque thing to produce incorrect output and keep trucking along in the face of errors. There are good reasons why PHP is the exception in doing this. – **Joel** Aug 18 '13 at 16:14

4 +1 for the BASIC comment. I don't advise using `NaN` in a beginner's language, but in general, great answer. – **Ross Patterson** Aug 19 '13 at 11:15

8 @Joel If he'd lived long enough, Dijkstra would probably have said "The use of [PHP] cripples the mind; its teaching should, therefore, be regarded as a criminal offense." – **Ross Patterson** Aug 19 '13 at 11:16

11 @Ross. "arrogance in computer science is measured in nano-Dijkstras" -- Alan Kay – **Jean-Claude Arbaut** Aug 19 '13 at 12:44

1 - Ignore the error and produce `0` as the result. Logging a warning if possible.

That's not a good idea. At all. People will start depending on it and should you ever fix it, you will break a lot of code.

2 - Add `NaN` as a possible value for numbers, but that raises questions about how to handle `NaN` values in other areas of the language.

You should handle NaN the way runtimes of other languages do it: Any further calculation also yields NaN and every comparison (even `NaN == NaN`) yields false.

I think this is acceptable, but not necessarily new comer friendly.


3 - Terminate the execution of the program and report to the user a severe error occurred.




This is the best solution I think. With that information in hand users should be able to handle 0. You should provide a testing environment, especially if it's intended to run once a night.

There's also a fourth option. Make division a ternary operation. Any of these two will work:

- `div(numerator, denominator, alternative_result)`
- `div(numerator, denominator, alternative_denominator)`

answered Aug 18 '13 at 13:43

 **back2dos**

21.4k  1  45  87

2 @AJMansfield: Either that, or people implement it themselves: `isNaN(x) => x != x`. Still, when you have `NaN` coming up in your programming code, you should not start adding `isNaN` checks, but rather track down the cause and make the necessary checks there. Therefore it is important for `NaN` to propagate fully. – **back2dos** Aug 18 '13 at 21:31

4 `NaN` s are majorly counter-intuitive. In a beginner's language, they're dead on arrival. – **Ross Patterson** Aug 19 '13 at 11:17

1 @RossPatterson But a beginner can easily say `1/0` -- you have to do something with it. There is no possibly useful result other than `Inf` or `NaN` -- something that will propagate the error further into the program. Otherwise the only solution is to stop with an error at this point. – **Mark Hurd** Aug 21 '13 at 1:23

1 Option 4 could be improved by allowing the invocation of a function, which in turn could perform whatever action is necessary to recover from the unexpected 0 divisor. – **CyberFonic** Jul 14 at 6:41

Terminate the running application with extreme prejudice. (While providing adequate debug information)

Then educate your users to identify and handle conditions where the divisor might be zero (user entered values, etc.)

edited Aug 18 '13 at 13:29

answered Aug 18 '13 at 13:23

 **DaveNay**
3,018 🟡 2 ⚪ 9 🟠 22

Other answers have already considered the relative merits of your ideas. I propose another one: use basic flow analysis to determine whether a variable *can* be zero. Then you can simply disallow division by variables that are potentially zero.

```
x = ...
y = ...

if y ≠ 0:
    return x / y    // In this block, y is known to be nonzero.
else:
    return x / y    // This, however, is a compile-time error.
```

Alternatively, have an intelligent assert function that establishes invariants:

```
x = ...
require x ≠ 0, "Unexpected zero in calculation"
// For the remainder of this scope, x is known to be nonzero.
```

This is as good as throwing a runtime error—you skirt undefined operations entirely—but has the advantage that the code path need not even be hit for the potential failure to be exposed. It can be done much like ordinary typechecking, by evaluating all branches of a program with nested typing environments for tracking and verifying invariants:

```
x = ...           // env1 = { x :: int }
y = ...           // env2 = env1 + { y :: int }
if y ≠ 0:         // env3 = env2 + { y ≠ 0 }
    return x / y   // (/) :: (int, int ≠ 0) → int
else:             // env4 = env2 + { y = 0 }
    ...
...              // env5 = env2
```

Furthermore, it extends naturally to range and `null` checking, if your language has such features.

edited Aug 18 '13 at 21:33

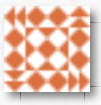
answered Aug 18 '13 at 21:23

 **Jon Purdy**
10.8k 🟡 2 ⚪ 34 🟠 75

- 3 Neat idea, but this type of constraint solving is NP-complete. Imagine something like `def foo(a,b): return a / ord(sha1(b)[0])` . The static analyzer can't invert SHA-1. Clang has this type of static analysis and its great for finding shallow bugs but there are plenty of cases it can't handle. — **mehaase** Jul 12 at 20:12 ~
- 6 this is not NP-complete, this is impossible -- says halting lemma. However, the static analyzer doesn't need to solve this, it can just crap out on a statement like this and require you to add an explicit assertion or decoration. — **MK01** Jul 13 at 4:07

In Haskell (and similar in Scala), instead of throwing exceptions (or returning null references) the wrapper types `Maybe` and `Either` can be used. With `Maybe` the user has a chance to test if the value he got is "empty", or he might provide a default value when "unwrapping". `Either` is similar, but can be used returns an object (e.g. an error string) describing the problem if there is one.

answered Aug 18 '13 at 16:34

 **Lande**
1,635 ⚪ 5 🟠 11

1 True, but note that Haskell doesn't use this for division by zero. Instead, every Haskell type implicitly has "bottom" as a possible value. This isn't like null pointers in the sense that it's the "value" of an expression that fails to terminate. You can't test for nontermination as a value, of course, but in operational semantics the cases that fail to terminate are part of the meaning of an expression. In Haskell, that "bottom" value also handles additional error-case results such as the `error "some message"` function being evaluated. — Steve314 Nov 15 '13 at 16:46

Number 1 (insert undebuggable zero) is always bad. The choice between #2 (propagate NaN) and #3 (kill the process) depends on context and ideally should be a global setting, as it is in Numpy.

If you're doing one big, integrated calculation, propagating NaN is a bad idea because it will eventually spread and infect your whole calculation--- when you look at the results in the morning and see that they're all NaN, you'd have to throw out the results and start again anyway. It would have been better if the program terminated, you got a call in the middle of the night and fixed it--- in terms of the number of wasted hours, at least.

If you're doing many little, mostly-independent calculations (like map-reduce or embarrassingly parallel calculations), and you can tolerate some percentage of them being unusable due to NaNs, the that's probably the best option. Terminating the program and not doing the 99% that would be good and useful on account of the 1% that are malformed and divide by zero might be a mistake.

Another option, related to NaNs: the same IEEE floating-point specification defines Inf and -Inf, and these are propagated differently than NaN. For instance, I'm pretty sure that $\text{Inf} > \text{any number}$ and $-\text{Inf} < \text{any number}$, which would be what you wanted if your division by zero happened because the zero was just supposed to be a small number. If your inputs are rounded and suffer from measument error (like physical measurements taken by hand), the difference of two large quantities can result in zero. Without the division-by-zero, you would have gotten some large number, and maybe you don't care how large it is. In that case, In and -Inf are perrfectly valid results.

It can be formally correct, too--- just say you're working in the extended reals.

answered Aug 18 '13 at 20:33

 Jim Pivarski

201 1 3

3. *Terminate the execution of the program and report to the user a severe error occurred.*

[This option] is not practical...

Of course it is practical: It is the programmers' responsibility to write a program that actually makes sense. Dividing by 0 does not make any sense. Therefore, if the programmer is performing a division, it is also his/her responsibility to verify *beforehand* that the divisor is not equal to 0. If the programmer fails to perform that validation check, then s/he should realize that mistake as soon as possible, and denormalized (NaN) or incorrect (0) computation results simply won't help in that respect.

Option 3 happens to be the one I would have recommended to you, btw, for being the most straightforward, honest, and mathematically correct one.

answered Aug 19 '13 at 9:11

community wiki
stakx

It seems like a bad idea to me to run important tasks (ie; "nightly cron") in an environment where errors are ignored. It's a terrible idea to make this a feature. This rules out options 1 and 2.

Option 3 is the only acceptable solution. Exceptions don't have to be part of the language, but they are part of reality. Your termination message ought to be as specific and informative as possible about the error.

answered Aug 21 '13 at 19:08

 ddyer

2,972 4 11

I think the problem is " targeted at novice users. --> So there is no support for ..."

Why are you thinking that exception handling is problematic for novice users?

What is worse? Have a "difficult" feature or have no idea why something happened? What could confuse more? A crash with a core dump or "Fatal error: Divide by Zero"?

Instead, I think is FAR better to aim for GREAT message errors. So do instead: "Bad calculation, Divide 0 / 0" (ie: Always show the DATA that cause the problem, not just the *kind* of problem). Look at how PostgreSql do the message errors, that are great IMHO.

However, you can look at other ways to work with exceptions like:

<http://dlang.org/exception-safe.html>

I also have dream on build a language, and in this case I think that mix a Maybe/Optional with normal Exceptions could do be the best:

```
def openFile(fileName): File | Exception
    if not(File.Exist(fileName)):
        raise FileNotExist(fileName)

    else:
        return File.Open()

#This cause a exception:

theFile = openFile('not exist')

# But this, not:

theFile | err = openFile('not exist')
```

answered Jul 13 at 4:45



mamcx

192 🍊 6

IEEE 754 actually has a well defined solution for your problem. Exception handling without using exceptions http://en.wikipedia.org/wiki/IEEE_floating_point#Exception_handling

```
1/0 = Inf
-1/0 = -Inf
0/0 = NaN
```

this way all your operations make mathematically sense.

$$\lim_{x \rightarrow 0} 1/x = \text{Inf}$$

In my opinion following IEEE 754 makes the most sense since it ensures that your calculations are as correct as it going to be on a computer and you are also consistent with the behaviour of other programming languages.

The only problem that arises is that Inf and NaN are gonna contaminate your results and your users will not know exactly where the problem is coming from. Take a look at a language like Julia that does this quite well.

```
julia> 1/0
Inf

julia> -1/0
-Inf

julia> 0/0
NaN

julia> a = [1,1,1] ./ [2,1,0]
3-element Array{Float64,1}:
 0.5
 1.0
 Inf

julia> sum(a)
```



```
Inf

julia> a = [1,1,0] ./ [2,1,0]
3-element Array{Float64,1}:
 0.5
 1.0
 NaN

julia> sum(a)
NaN
```

The division error is correctly propagated through the mathematical operations but at the end the user does not necessarily know from which operation the error stems from.

edit: I didn't see the second part of the answer by Jim Pivarski which is basically what I am saying above. My bad.

answered Jul 13 at 7:29

 **wallnuss**
121 🟡 2

In my view your language should provide a generic mechanism for detecting and handling errors. Programming errors should be detected at compile time (or as early as possible) and should ordinarily lead to program termination. Errors that result from unexpected or erroneous data, or from unexpected external conditions, should be detected and made available for appropriate action, but allow the program to continue whenever possible.

Plausible actions include (a) terminate (b) prompt the user for an action (c) log the error (d) substitute a corrected value (e) set an indicator to be tested in code (f) invoke an error handling routine. Which of these you make available and by what means are choices you have to make.

From my experience, common data errors like faulty conversions, divide by zero, overflow and value out of range are benign and should by default be handled by substituting a different value and setting an error flag. The (non-programmer) using this language will see the faulty data and quickly understand the need to check for errors and handle them.

[For an example, consider an Excel spreadsheet. Excel does not terminate your spreadsheet because a number overflowed or whatever. The cell gets a strange value and you go find out why and fix it.]

So to answer your question: you should certainly not terminate. You might substitute NaN but you should not make that visible, just make sure the calculation completes and generates a strange high value. And set an error flag so that users who need it can determine that an error occurred.


Disclosure: I created just such a language implementation (Powerflex) and addressed exactly this problem (and many others) in the 1980s. There has been little or no progress on languages for non-programmers in the last 20 years or so, and you will attract heaps of criticism for trying, but I really hope you succeed.

answered Jul 12 at 11:18

 **david.pfx**
5,110 🟡 3 🟡 25

SQL, easily the language most widely used by non-programmers, does #3, for whatever that's worth. In my experience observing and assisting non-programmers writing SQL, this behavior is generally well understood and easily compensated for (with a case statement or the like). It helps that the error message you get tends to be quite direct e.g. in Postgres 9 you get "ERROR: division by zero".

answered Jul 13 at 4:28

 **Noah Yetter**
111 🟡 2


I liked the ternary operator where you provide an alternate value in case the denominator is 0.

One more idea I didn't see is to produce a general "invalid" value. A general "this variable doesn't have a value because the program did something bad", which carries a full stack trace with itself. Then, if you ever use that value anywhere, the result is again invalid, with the new operation attempted on top (i.e. if the invalid value ever appears in an expression, the entire expression yields invalid and no function calls are attempted; an exception would be boolean operators - true or invalid

valid and no function calls are attempted, an exception would be boolean operators - true or invalid is true and false and invalid is false - there may be other exceptions to that, too). Once that value is no longer referenced anywhere, you log a nice long description of the entire chain where things were wrong and continue business as usual. Maybe email the trace to the project lead or something.

Something like the Maybe monad basically. It will work with anything else that can fail, too, and you can allow people to construct their own invalids. And the program will continue to run as long as the error isn't too deep, which is what is really wanted here, I think.

answered Jul 13 at 7:05

 **Moshev**
11 1

Disallow it in the language. That is to say, disallow dividing by a number until it's provably not zero, usually by testing it first. Ie.

```
int div = random(0,100);
int b = 10000 / div; // Error E0000: div might be zero
```

answered Nov 15 '13 at 16:52

 **MSalters**
4,004 5 20

2 But only in certain cases; it can't do so, with 100% accuracy, in all cases. You'll either have false positives or false negatives. This is provably true. For example, I could create a snippet of code that may or may not even *complete*. If the compiler can't even know if it finished, how could it know if the resulting int is non-zero? It can catch simple obvious cases, but not *all* cases. – Servy Nov 15 '13 at 18:35

There are two fundamental reasons for a divide by zero.


1. In a precise model (like integers), you get a divide by zero DBZ because the input is wrong. This is the kind of DBZ that most of us think of.
2. In non-precise model (like floating pt), you might get a DBZ because of rounding-error even though the input is valid. This is what we don't normally think of.

For 1. you have to communicate back to the users that they made a mistake because they are the one responsible and they are the one who know best how to remedy the situation.

For 2. This is not user fault, you can finger point to algorithm, hardware implementation, etc but this is not user's fault so you shouldn't terminate the program or even throw exception (if allowed which is not in this case). So a reasonable solution is to continue the operations in some reasonable manner.

I can see the the person asking this question asked for case 1. So you need to communicate back to the user. Using whatever floating point standard, Inf, -Inf, Nan, IEEE doesn't fit in this situation. Fundamentally wrong strategy.

answered Jul 13 at 8:11


 **randomA**
928 1 13

As you a writing a programming language, you should take advantage of the fact and make it mandatory to include an action for the devise by zero state. $a \leq n / c : 0 \text{ div-by-zero-action}$

I know what I've just suggested is essentially adding a 'goto' to your PL.

answered Jul 13 at 19:02

 **Stephen**
179 2 8

 **protected by Community ♦ Jul 13 at 7:29**
Thank you for your interest in this question. Because it has attracted low-quality answers, posting an answer now requires 10 reputation on this site.

Would you like to answer one of these [unanswered questions](#) instead?

