Three ways to report failure

Computational functions often fail. When one does, it must report failure in some way. To misquote a book and movie about the Apollo 13 almost disaster (clickable link): Failure is inevitable, but *silent failure is not an option.* A function that can fail should report that failure and the calling code should look for the report and handle it in some way.

The Python code `exceptionDemo.py` posted with Assignment 9 explains three ways this can be done. The task in the demo is computing $y = \sqrt{x}$. This fails only if $x < 0$. The code for computing $\sqrt{x}$ should return $\sqrt{x}$ if $x \geq 0$. Otherwise, it should communicate to the calling code that it could not compute $\sqrt{x}$. The calling code should look for the report and handle the error in some way. In this case, it either prints $\sqrt{x}$, or, if $x < 0$, prints something saying it did not compute $\sqrt{x}$.

In more realistic situations, the error/failure could be that a data file does not exist or has the wrong format (two different kinds of failure). An iterative algorithm like gradient descent could have iterates that fail to converge. A function may receive arguments that are not feasible or violate some constraints, such as $\sqrt{x}$ with $x < 0$. A matrix factorization could fail to exist, such as Cholesky, $H = LL^T$, when H has negative eigenvalues.

The first method is illustrated in the function `myRootNaN` defined starting on line 16. The IEEE floating point standard says that the square root function should return `NaN` if the argument is negative. The `numpy` function `isnan()` returns `True` if its argument is `NaN` and `False` otherwise. The function `myRootNan` is called on line 44 and line 45 tests whether it was successful by checking whether it got `NaN` or not. This is the simplest approach to the $x < 0$ problem, but it has the drawback that Python doesn't like making `NaN` values, so it prints a warning that this has happened. The warning might seem harmless, but it interferes with the carefully formatted output. You would not want your boss to see the warning; it looks unprofessional.

The second approach is illustrated in the function `myRootNone` defined starting on line 23. This avoids generating a `NaN` by not calling `np.sqrt(x)` if $x < 0$. If $x < 0$, the function returns `None`, which is the Python keyword for the "null object", an object with no content. In this case, binds `y` to a float if $x \geq 0$ and binds `y` to the null object otherwise. Line 56 determines whether the function failed by testing whether or not `y` has been bound to the null object. This approach is better than the first one because it avoids the `NaN` warning in the output.

The third approach is the one professionals prefer, as you can confirm by doing a web search on: `python error handling best practices`. The function `myRootExcept` (starting on line 34) *raises an exception.*[1] The keyword `try`

[1] This would be called *throwing* an exception that the calling code should *catch* in C++

tells Python to look for an exception. The keyword `except` tells Python to test whether an exception was raised. The code in that block (lines 69 and 70) says what to do if the exception was raised. The `else` branch (lines 72 and 73) says what to do if the exception was not raised. The `finally` keyword starts a block of code to be executed whether or not the exception was raised.

In Python, an *exception* is an instance of an exception class. An exception class is any class that is a *derived class* from the *base class* `Exception`.[2] You can read about base and derived classes in the Python documentation: Section 9.5, *Inheritance* (clickable link). Line 31 defines `NegativeArgument` as the name of a derived class from the base class `Exception`. The derived class *inherits* (has) all the variable and function definitions in the base class. This means that the new derived class `NegativeArgument` can do anything the base class `Exception` can do, which is everything that it needs to do to be a Python exception. The documentation (a bit cryptic at first) gives a long list for specialists, see Section 9.5, *Inheritance* (clickable link). Everything in the constructor of the base class is inherited by the derived class. If you don't need the derived class to do anything the base class can't do, you can just end the derived class definition with `pass`, which is line 32.

The keyword `raise` on line 36 calls the constructor for the `NegativeArgument` class to instantiate (create) an object of that class. This explains the parens after `NegativeArgument`. This class instance does not get a name, but the *exception handler*, which is part of the Python interpreter, keeps a pointer to it. The keyword `try` on line 66 tells the interpreter to execute the indented code block with the exception handler turned on. The keyword `except` on line 68 asks the exception handler for any exceptions it might have. Remember, an exception is an instance of a class that is derived from the `Exception` class. The `NegativeArgument` on line 68 asks the exception handler for any exceptions that are instances of that class.[3] If there are any, the interpreter executes lines 69 and 70. If there are none, the interpreter executes lines 72 and 73. Either way, it executes the `finally` code block, which is line 75.

There is more to exception handling than just this. There is a mechanism for the exception instance to have information about why the exception was raised and communicate this to the exception handler.

---

or Java. The equivalent of lines 66 to 75 would be called a *try catch* block.

[2] Python convention is that the name of a class starts with a capital letter.

[3] If line 68 has just been `except:` (not naming the specific exception class you're looking for), that would have been a *naked* exception handler. Python documentation tells you not to do this, because other functions might have raised other exceptions not related to what you're doing here.