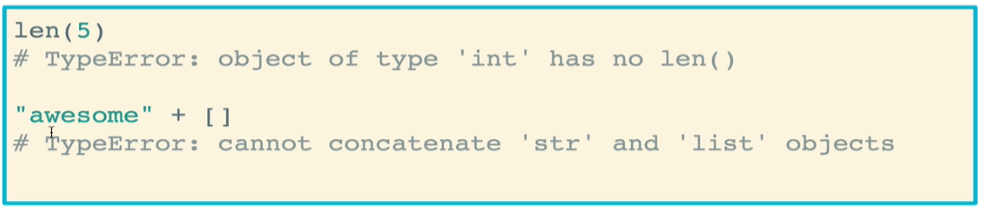
* Errors and mistakes are very common in Python code. Understanding errors is key to knowing how to fix them
* The good old **SyntaxError** occurs when Python encounters incorrect syntax (something it doesn’t parse)
  + Usually due to typos or not knowing the syntax of Python well enough



* **NameError** occurs when a variable has not been defined, i.e. has not been assigned yet
  + If a variable does not exist and you try to call it, you will be treated to a NameError



* **TypeError** occurs when an operation or function is applied to the wrong type of data. In this case, Python cannot interpret an operation on two data types
  + Example: integers and floats have no len() property
  + Example: string and list objects, though both iterables, cannot be concatenated with one-another



* **IndexError** occurs when you try to access an element in a list using an invalid index
  + An index that is out of the range of the list or string



* **ValueError** occurs when a built-in function receives an argument that has the right type but an inappropriate value
  + Example: You are able to convert a string to an integer only if that string is a number



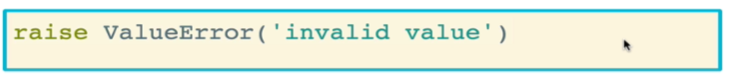
* **KeyError** occurs when you attempt to access a key within a dictionary does not exist
  + Think of it as an IndexError for dictionaries



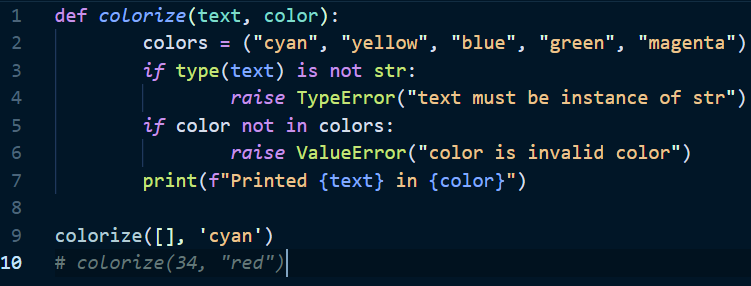
* **AttributeError** occurs when a variable does not have an attribute that you are trying to access
  + Example: trying to apply the *foo* method for a string. Strings have no *foo* attribute



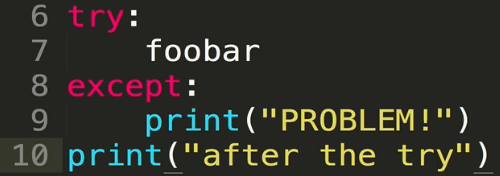
* Additional errors can be found in the Python documentation: <https://docs.python.org/3/library/exceptions.html>
* You can also raise your own error using the keyword **raise**
  + You can select the type of error to raise, as well as a custom message describing the error
  + It is up to you what error to raise and how to raise it. However it is helpful to raise errors that are pertinent to what you are doing
    - Do not raise a divide by zero error for an operation that has nothing to do with mathematical division
    - The custom message should provide the user useful information for how to correct the error



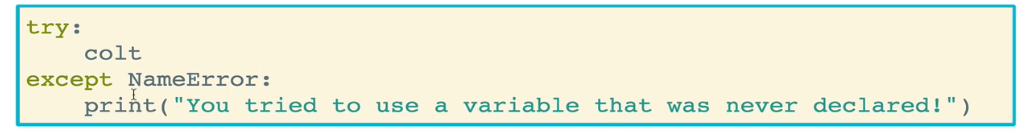
* + Example: a colorize function that checks whether the text being colorized is actually a string, and whether the color selected is in the list of valid colors
    - Note that this function does not actually color any text. It just helps us illustrate the point of raising errors



* We use **try and except blocks** to actually handle errors when we ancounter them
  + In Python, it is strongly encouraged to use try/except blocks to catch exceptions when we can do something about them
  + Example: this code will attempt to call *foobar* which is undefined. By using the except block, we can print that there is a problem and continue running the code



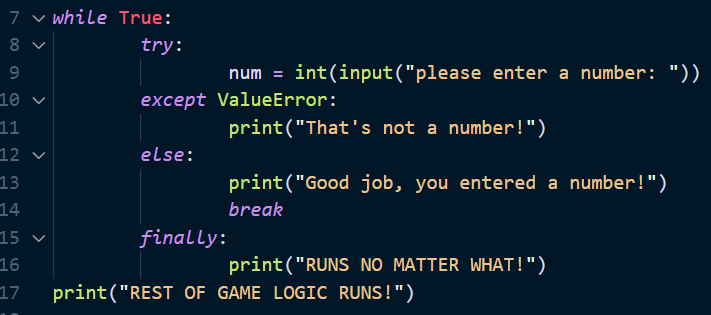
* + It is a good idea to use particular errors with *except*. This will ensure that the except block will only run if the indicated type of error is encountered



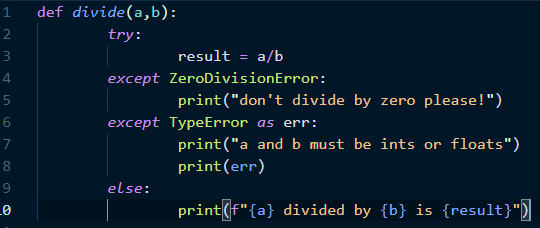
* + Example: we will specifically check for a KeyError when trying to access a key in this dictionary
    - Don’t confuse this with the “get” method for dictionaries which retrieves the value for a g



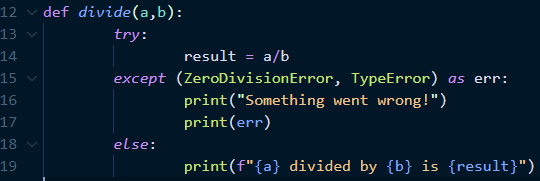
* The try and except blocks can also use **else** and **finally**, and are used less commonly than try and except
  + **Else** will run when except does not run (basically when try is successful)
    - Else clauses typically include a break to exit the loop
  + **Finally** will run no matter what
  + Example: skeleton code for a game that asks for the user top enter a number. This try and except block will continue to run until the user enters a valid number



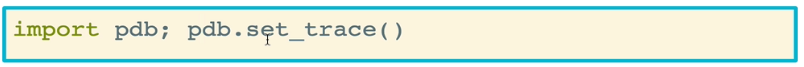
* + Example: This function will cover different types of errors that can be excepted
    - You can have multiple except clauses, one capturing a different type of error
      * Note that you can call the error type *as* a variable (usually called **err**), and then print that error. This gives the user the official error feedback provided by Python



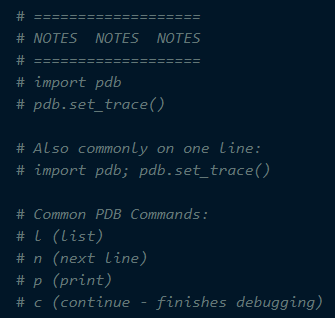
* + - Instead of having multiple except clauses, you can also combine the multiple error types into a single tuple under one *except* clause
      * The drawback here is that you will need to be more generic in your error message, since different types of error can occur.
      * To skirt this issue, you can pass in the error type *as* a variable (usually **err**) once again, and then print that error so that the user knows which error was encountered
      * You can also use *if* statements on **err** to check what type of error was encountered, and then modify the message accordingly. However, this is similar to just having different *except* clauses, and so the instructor actually prefers the latter.



* The **Python debugger** (**pdb**) helps you deal with errors in your code that you do not expect
  + A loud error breaks your code and reports the issue
  + A silent error (or silent bug) is when your code doesn’t break, but it doesn’t work the way you expect (super frustrating)
  + It is a module that must be imported to be used
    - You insert **pdb.set\_trace()**, which will allow you to step through what is going on. This command tells Python to pause, which allows you to investigate in the terminal
    - Note: You can insert pdb.set\_trace() anywhere, but wherever you call it, that’s where the debugging trace mode will start. Instructor usually places it shortly before the lines that are breaking



* + The **pdb.set\_trace()** comes with command also comes with a list of commands that can be used to debug further



* + - *l*, or list, tells you where you are in the execution, and an error will show you what is going to happen next
    - This also allows you to explore the variable space, determinine which variables that are or are not defined
    - You can then use *n* to move to the next line to execute that line, and then use *l* again to see where you are and check which values have been updated
    - The *c* command allows you to exit debugging and continue running the code
  + Important gotcha note: if you have variable names that conflict with the pdb commands (e.g. l, n, p, c), you will run into issues exploring those variables while in debug modes
    - The best way to avoid this is to not use single-letter variable names ever, let alone ones that are conflicting with pdb commands
    - If you must use such variable names, then you can use the *p* command prior to calling the variable to explore its value