Unit 3-3: Error Handling

* Python Error Defined:
  + - Noun
    - An calamitous situation in which a large serpent ties itself into a not
    - A programmer oopsie; the inevitable
  + As you continue programming with Python, you'll no doubt encounter a variety of errors fed back to you by the interpreter. Having your program halt with an error message indicates that something is wrong with the code. Learning to understand these messages — and what to do next — is an essential part of becoming a proficient programmer.
  + There are many different types of errors and exceptions in Python. Their names (for the most part) provide insight into the reason why the code failed to run. In this lesson, we'll review a variety of errors and exceptions that exemplify this behavior.
* NameError
  + One basic type of error is a NameError. A NameError will be thrown when the variable that's referred to doesn't exist in the name space — in other words, when we have yet to define the variable but are trying to access it.
    - In [1]: hello
    - ---------------------------------------------------------------------------
    - NameError Traceback (most recent call last)
    - <ipython-input-1-b1946ac92492> in <module>()
    - ----> 1 hello
    - NameError: name 'hello' is not defined
  + Of course, if we did have a variable named hello, this error wouldn't come up.
    - In [2]: hello = True
    - In [3]: hello
    - Out[3]: True
* SyntaxError
  + SyntaxError is arguably the most common type of error. A SyntaxError indicates that the logic of the code is incorrect or contains a typo. For example, if we failed to place a colon after an if statement condition, we'd see something like this:
    - In [4]: if hello print('hello exists')
    - File "<ipython-input-4-3bd228f97f89>", line 1
    - if hello print('hello exists')
    - ^
    - SyntaxError: invalid syntax
  + The fix? Simply put the colon in its correct place.
    - In [6]: if hello: print ('hello exists')
    - hello exists
  + Take a look at the SyntaxError again. This error provides a user with some information to help diagnose the problem:
    - In [4]: if hello print('hello exists')
    - File "<ipython-input-4-3bd228f97f89>", line 1
    - if hello print('hello exists')
    - ^
    - SyntaxError: invalid syntax
  + An arrow (^) under the offending section of code indicates where the problem occurred. In this case, the fix comes in earlier — we need to put a colon after the hello; however, the Python interpreter fails on print.
  + The error itself contains a vague message: invalid syntax. This particular SyntaxError message isn't useful, but sometimes the explanation following an error will contain context that's helpful for diagnosing the problem.
* TypeError
  + TypeError occurs when we try to add two different types of objects together:
    - In [7]: 12.0 + 'twelve'
    - ---------------------------------------------------------------------------
    - TypeError Traceback (most recent call last)
    - <ipython-input-7-0525c126347b> in <module>()
    - ----> 1 12.0 + 'twelve'
    - TypeError: unsupported operand type(s) for +: 'float' and 'str'
  + The TypeError already tells us that the error is related to Python data types. In this case, the message is clear (although it does contain a fair amount of jargon) — you can't use the operand + with a float and a string.
* Handling Exceptions
  + There are times when we, as programmers, may want to allow an exception to occur. But, instead of having the exception stop the whole program, what if we could identify when one has occurred and take action accordingly?
  + Consider this example: We want to convert strings into numbers if possible and otherwise replace them with None. Say we have the following sample of strings to convert, some of which can be converted to floats and some of which can't.
    - * str\_to\_float = ['2.1', '2.3', '7,5', '$12.12', '8.9', '5%', '33.1']
    - Looking over the list, we determine that this logic will successfully accomplish what we need:
      * floats = []
      * bad\_chars = ',$%'
      * for s in str\_to\_float:
      * corrupted = False
      * for bc in bad\_chars:
      * if bc in s:
      * corrupted = True
      * if corrupted:
      * floats.append(None)
      * else:
      * floats.append(float(s))
    - Here, we review each string we want to convert. For each item in the list, we also check for all of the characters that would cause the float function to throw an error. We only convert the string to a float if none are detected, otherwise we append None to the list instead.
    - Once we've executed the code block floats should be [2.1, 2.3, None, None, 8.9, None, 33.1].
  + That said, we've only seen some of the strings. What if there were millions of different strings we wanted to convert? What if some of them had other characters that could break the float function?
  + It may be impossible to look through every element and come up with the logic to ensure the process doesn't throw an error. Even if it were possible, our code could quickly become bloated with condition after condition that needed to be met.
* Try-Except
  + Luckily, we can wait for float to break instead and then take an action. The basic syntax for handling exceptions uses the keywords try and except. Here's how we can use them given the example above:
    - floats = []
    - for s in str\_to\_float:
    - try:
    - floats.append(float(s))
    - except:
    - floats.append(None)
  + Just like before, we initialize an empty list to hold the converted numbers. Now, as we iterate through the str\_to\_float list, we use the try and except syntax to handle errors. First, try attempts the code inside of its block. If that code successfully runs, except is skipped. Alternatively, if the code in the try block throws an exception, the code inside of the except block will run instead.
    - try:
    - floats.append(float(s))
    - except:
    - floats.append(None)
  + The code above simply catches any exception that could occur in the try statement. It may be more useful to be specific about the type of error on which you want to act. Fortunately, except statements can be followed by the type of exception (or a list of types).
    - try:
    - # Code
    - except TypeError as target:
    - # Code to run on TypeError.
    - except (NameError, ValueError) as target:
    - # Code to run if name or value error thrown.
    - except:
    - # All other exceptions.
  + Splitting the except statement into multiple sections delineated by error type improves the flexibility of your code in the face of uncertain errors. Note that you can also chain multiple except statements together.
  + **In Python 2, the syntax for catching exceptions was except ExceptionType:, or except ExceptionType, target: when the exception object is desired. ExceptionType could be a tuple, as in, for example, except (TypeError, ValueError):. In Python 3, the syntax except ExceptionType as target: was introduced to avoid hard-to-spot bugs.**
* Try-Except-Else
  + Two other keywords can be used in try statements: else and finally.
  + The else statement follows any try and except statement code and will execute only if no exception has been thrown. For example, say we had a function that adds variables together:
    - def adder(x, y):
    - try:
    - z = x+y
    - except TypeError as target:
    - print ('cannot add these types together')
    - else:
    - print('result is:', z)
  + First we try to add x and y together and assign the result to a new variable, z. If a TypeError is thrown, we print out a message to the user stating that they cannot add these types together. If no TypeError is thrown, then the code in the else statement will run.
    - In [1]: adder(1, 3)
    - result is: 4
    - In [2]: adder('1', 3)
    - cannot add these types together
* Try-Except-Else-Finally
  + The finally statement comes at the end of the code and will always run — regardless of whether or not an exception is caught. We can add a finally statement to our adder() function to see the result:
    - def adder(x, y):
    - try:
    - z = x+y
    - except TypeError as target:
    - print ('cannot add these types together')
    - else:
    - print('result is:', z)
    - finally:
    - print('function ending.')
  + Now, when we run the same code, you can see that the finally statement will run either way:
    - In [10]: adder(1, 3)
    - result is: 4
    - function ending.
    - In [11]: adder('1', 3)
    - cannot add these types together
    - function ending.
* Raise
  + If you are writing code, it is easy to raise your own exceptions when specific conditions are met. The raise keyword can be used to alert a user of a specific error type. As an example, we can write a function that will raise an error if 4 is entered and otherwise print the number:
    - def no\_four(number):
    - if number == 4:
    - raise ValueError('No fours allowed!!')
    - else:
    - print('Number:', number)
  + The string entered as an argument to ValueError will be the message that's returned to a user when the error is raised:
    - In [20]: no\_four(4)
    - ---------------------------------------------------------------------------
    - ValueError Traceback (most recent call last)
    - <ipython-input-20-a01fa7d42066> in <module>()
    - ----> 1 no\_four(4)
    - <ipython-input-19-7adc614ec34c> in no\_four(number)
    - 1 def no\_four(number):
    - 2 if number == 4:
    - ----> 3 raise ValueError('No fours allowed!!')
    - 4 else:
    - 5 print 'Number:', number
    - ValueError: No fours allowed!!
    - In [21]: no\_four(2)
    - Number: 2
* A Best Practice
  + When you write your own code and maintain it for other users, it is a good practice to provide meaningful, clear error messages.
  + This introduction to error handling should help you not only debug your own code and understand Python's error messages, but eventually write your own exception logic into code that you build and maintain for others.