Errors
And How to Handle Them
There is a saying in computer science: “Garbage in, garbage out.”

Is this true, or is it just an excuse for bad programming?

Answer: Both.

Here’s what you want:

Can you do it?
Errors and exceptions

- Your program isn’t alone in the universe—it has to interact with things outside itself
  - Users typing things in
  - Data read from files
  - Calls to your functions from “the outside world”
  - Your calls to “someone else’s” functions
- It helps to think in terms of “my code” and “not my code”
  - An error is a mistake in your code, and it’s your fault
  - An exception is a mistake that isn’t in your code, and isn’t your fault, but that doesn’t mean you can ignore it
- This isn’t entirely accurate, and the distinction isn’t really all that clear, but it’s a reasonable simplification
Sanity checking

• While Python has some built-in features for dealing with various kinds of mistakes, there are some simple things you can do without them

• *Sanity checking* is simply testing whether values are reasonable

  • `age = int(input("What is your age? "))`
    
    `if age < 3 or age > 120:
        print("No you're not!")`
Errors

• An error is a bug in your code—fix it!

• There is no substitute for careful testing, but when errors do occur—
  • Putting in print statements (and later removing them, when the code works) is generally the most helpful
  • Using the debugger to step through your code is sometimes helpful
  • assert statements (covered next) can, on rare occasions, be helpful

• Other suggestions—
  • Take a break—do something else for a while
  • Explain your code to a friend
    • They don’t have to understand you, but they should pretend to listen
    • If you have no friends, explain your code to your dog, or to your teddy bear
    • Not to your cat—they really don’t listen
• **Comments** are useful to the human reader, but ignored by the computer

• **Assertions** are useful to the human reader, *and* their validity can be checked by the computer
  • Unfortunately, assertions are limited to statements that can be expressed by a boolean expression

• **Syntax:**
  
  ```
  assert boolean_expression
  
  or
  assert boolean_expression, message
  ```

• If the boolean expression is **False**, an AssertionError occurs

• The primary purpose of the `assert` statement is to inform the human reader that the following code can assume the assertion is true

• The `message` is hardly every required, but if used, should provide additional information

• Another purpose of the assert statement is to tell you when you are mistaken, and contrary to your expectations, the boolean expression is **False**
**assert** and require

• Some languages have both “assert” and “require”
  
  • **assert** *boolean_expression* says that, at this point in the code, I believe the *boolean_expression* to be **True**
  
  • **require** *boolean_expression* says that, in order for the following code to work correctly, the *boolean_expression* must be **True**

• Python has only **assert**, but since “assert” and “require” do essentially the same thing (signal that an error has occurred), we can implement “require” using **assert**

  • **def** require(b):
    
    assert b, "Requirement not met."

• Or we can simply use **assert** to mean **require**
Example

- `def first_asterisk(s):
  """Returns the index of the first '*' in a string."""
  require('*' in s)
  for index in range(0, len(s)):
    if s[index] == '*':
      break
  assert s[index] == '*'
  assert '*' not in s[:index]
  return index

- This is excessive, and I don’t recommend it as a general practice, but the occasional use of `assert` can be helpful
Someone else’s problem

- An *exception* is a mistake that you can detect, but not a bug you can fix
  - It is “someone else’s problem”
  - Later in the course we will talk about *classes*, and the phrase “someone else” will mean “some other class”
- When you detect such a mistake, you should *raise* (or *throw*) an *exception*
  - Syntax: `raise name_of_exception(message)`
  - The most general type of exception has the name *Exception*, so usually you would just say `raise Exception(message)`
- Used correctly, raising an exception immediately exits the function that it is in
def square_root(n):
    """Finds the square root of a non-negative number."""
    if n < 0:
        raise Exception("square_root called with "
                      + str(n))
    epsilon = 0.0000001
    guess = n / 2
    quotient = n / guess
    while abs(quotient - guess) > epsilon:
        guess = (guess + quotient) / 2
        quotient = n / guess
    return guess

Raising the exception says to the caller, “You’re doing it wrong!”
Flow of control

• When an exception is raised, the usual flow of control is disrupted
  • The exception must be *caught*
    • We will talk about how to do this shortly
  • Here’s what happens when a function throws an exception:
    • Control immediately returns to the function that called this one
      • If that function catches the exception, it executes whatever code is necessary to deal with the situation
      • If the function does not catch the exception, it returns to the function that called it, and so on up the line
    • If no function ever catches the exception, the program terminates with an error message
      • In other words, the exception “propagates up the call chain” until it is caught, or the system catches it and terminates the program
  • Exceptions are for dealing with errors, *not* for routine flow of control
Catching exceptions

- Syntax:
  ```python
  try:
      code that might result in an exception
  except type_of_exception_1:
      code to do something when exception 1 occurs
  except type_of_exception_2:
      code to do something when exception 2 occurs
  finally:
      code to be performed whether or not an exception occurred
  ```

- You may have as many `except` clauses as you like
- `except` and `finally` are both optional, but you must have at least one
- Semantics:
  - The code in the `try` part is executed
  - If an exception occurs, the first `except` clause of the right type is executed
    - If no `except` clause is of the right type, control goes to the calling function
    - To catch *any* type of exception, say `except Exception:
  - The `finally` clause is executed, whether or not an exception occurred
Reading in an integer

- `def get_int(prompt):
  """Gets an integer from the user"""
  try:
      age = int(input(prompt))
  return age
  except ValueError:
      print("That's not an integer!")
  return get_int(prompt)

- `>>> get_int("What is your age? ")
  What is your age? fh
  That's not an integer!
  What is your age? dgghdf
  That's not an integer!
  What is your age? 99
  99  (result printed by IDLE)`
Getting the message

- **try:**
  
  ```python
  root = square_root(-5)
  except Exception as msg:
      print(msg)
  ```

- **square_root called with -5**

- If the exception occurs and you don’t catch it, the program will crash and the system will print out the message

- ```
  >>> square_root(-5)
  Traceback (most recent call last):
      File "<pyshell#42>" , line 1, in <module>
        square_root(-5)
      File "/Users/dave/Box Sync/Programming/Python3_programs/scratch.py", line 34, in square_root
         raise Exception("square_root called with " +
      str(n))
  Exception: square_root called with -5
  ```
assert or raise?

• When an `assert` statement fails, it raises an `AssertionError`, which is a kind of `Exception`.
• Hence, the following two things are almost exactly equivalent:
  • `assert boolean_expression, message`
  • `if not boolean_expression: raise AssertionError(message)`
• So when do you use which?
  • You should `assert` things which you really expect to be true; and if they aren’t, you should fix your code so that they are
  • When your function gets invalid data (provided by “someone else”) you should `raise` an exception, thus making it “their problem”
“Never test for an error condition you don’t know how to handle.”

- Steinbach’s Guideline for Systems Programming