ERROR HANDLING

LECTURE 12–2

JIM FIX, REED COLLEGE CSCI 121

COURSE INFO

- **Today:** error handling; **Roll100** tournament
- **Homework 12:** on BSTs, files, errors, **due May 11.**
- **Project 4:** a text-based role-playing game, **due May 11th.**
  - Checkpoint of 15-20 points is due **Friday!**
- **Next Monday:** second midterm exam. Topics covered:
  - recursion
  - object orientation
  - inheritance
  - higher-order functions
  - linked lists
- I've posted a practice exam and will post its solutions **Friday**.

SIMPLE INPUT PROGRAM

- Consider this simple script that gets an input and computes with it:

```
import math
x = int(input("Enter a positive integer: "))
y = 100 // math.isqrt(x)
print("100 / sqrt(" + str(x) + ") is " + str(y) + ".")
```

- What could happen?

SIMPLE INPUT PROGRAM

- Consider this simple script that gets an input and computes with it:

```
import math
x = int(input("Enter a positive integer: "))
y = 100 // math.isqrt(x)
print("100 / sqrt(" + str(x) + ") is " + str(y) + ".")
```

- What could happen?
  - They enter a positive integer, all is good.

```
$ python3 sample_input_script.py
Enter a positive integer: 25
100 / sqrt(25) is 20.
$
```

SIMPLE INPUT PROGRAM

- Consider this simple script that gets an input and computes with it:

```
import math
x = int(input("Enter a positive integer: "))
y = 100 // math.isqrt(x)
print("100 / sqrt(" + str(x) + ") is " + str(y) + ".")
```

- What could happen?
  - They enter zero, and so we divide by 0. ***ERROR!***

```
$ python3 sample_input_script.py
Enter a positive integer: 0
Traceback (most recent call last):
  File "simple_input_script.py", line 3, in <module>
    y = 100 // math.isqrt(x)
        ~~~~^^~~~~~~~~~~~~~~
ZeroDivisionError: integer division or modulo by zero
$
```

SIMPLE INPUT PROGRAM

- Consider this simple script that gets an input and computes with it:

```
import math
x = int(input("Enter a positive integer: "))
y = 100 // math.isqrt(x)
print("100 / sqrt(" + str(x) + ") is " + str(y) + ".")
```

- What could happen?
  - ➡ They enter a negative integer which has no square root. ***ERROR!***

```
$ python3 sample_input_script.py
Enter a positive integer: -25
Traceback (most recent call last):
  File "simple_input_script.py", line 3, in <module>
    y = 100 // math.isqrt(x)
                ^^^^^^^^^^^^
ValueError: isqrt() argument must be nonnegative
$
```

SIMPLE INPUT PROGRAM

- Consider this simple script that gets an input and computes with it:

```
import math
x = int(input("Enter a positive integer: "))
y = 100 // math.isqrt(x)
print("100 / sqrt(" + str(x) + ") is " + str(y) + ".")
```

- What could happen?
  - ➡ They enter something with a typo. ***ERROR!***

```
$ python3 sample_input_script.py
Enter a positive integer: 25oops
Traceback (most recent call last):
  File "simple_input_script.py", line 2, in <module>
    x = int(input("Enter an integer: "))
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
ValueError: invalid literal for int() with base 10: '25oops'
$
```

SIMPLE INPUT PROGRAM

- Consider this simple script that gets an input and computes with it:

```
import math
x = int(input("Enter a positive integer: "))
y = 100 // math.isqrt(x)
print("100 / sqrt(" + str(x) + ") is " + str(y) + ".")
```

- What could happen?
  - ➡ The entry leads to some error, the program halts ("crashes").
- Can we write the code to anticipate and handle these errors?
  - ➡ One option: Hand-code input of `x` to do careful checks.

SIMPLE INPUT PROGRAM

- Consider this simple script that gets an input and computes with it:

```
import math
x = int(input("Enter a positive integer: "))
y = 100 // math.isqrt(x)
print("100 / sqrt(" + str(x) + ") is " + str(y) + ".")
```

- What could happen?
  - The entry leads to some error, the program halts ("crashes").
- Can we write the code to anticipate and handle these errors?
  - One option: Hand-code input of `x` to do careful checks.
  - An alternative: let errors happen, catch them and handle them.

CAREFULLY CHECKING OR PREVENTING AN ERROR

- This code avoids the errors by checking and looping:

```
while True:
    s = input("Enter a positive integer: ")
    if isdigit(s) and int(s) > 0:
        x = int(s)
        break
    print("Bad entry. Please try again.")

y = 100 // math.isqrt(x)
print("100 / sqrt(" + str(x) + ") is " + str(y) + ".")
```

CATCHING ERRORS WITH TRY... EXCEPT

- An alternative is to let errors happen. Use `try` block with `except` clause:

```
while True:
    try:
        x = int(input("Enter a positive integer: "))
        y = 100 // math.isqrt(x)
        break
    except:
        print("Bad entry. Please try again.")

print("100 / sqrt(" + str(x) + ") is " + str(y) + ".")
```

CATCHING ERRORS WITH TRY… EXCEPT

- This loops using a `try` block with an `except` clause:

```
while True:
    try:
        x = int(input("Enter a positive integer: "))
        y = 100 // math.isqrt(x)
        break
    except:
        print("Bad entry. Please try again.")

print("100 / sqrt(" + str(x) + ") is " + str(y) + ".")
```

- How does Python execute this?
  - ➡ It executes the lines in the `try` block.
  - ➡ If no error occurs, runs as normal (skipping the code under `except`).

CATCHING ERRORS WITH TRY… EXCEPT

- This loops using a **`try`** block with an **`except`** clause:

```
while True:
    try:
        x = int(input("Enter a positive integer: "))
        y = 100 // math.isqrt(x)
        break
    except:
        print("Bad entry. Please try again.")

print("100 / sqrt(" + str(x) + ") is " + str(y) + ".")
```

- How does Python execute this?
  - If an error occurs within the **`try`** block, execution of it halts.
  - Python then jumps down to the lines in the **`except`** clause.

CATCHING ERRORS WITH TRY… EXCEPT

- This loops using a `try` block with an `except` clause:

```
while True:
    try:
        x = int(input("Enter a positive integer: "))
        y = 100 // math.isqrt(x)
        break
    except:
        print("Bad entry. Please try again.")

print("100 / sqrt(" + str(x) + ") is " + str(y) + ".")
```

- So… code loops until input of `x` and calculation of `y` succeeds.

ERROR CHECKING

- Python `try… except` gives us a different style of coding.
- Sometimes cleaner to let errors happen wherever within code's inner layers, handle them in appropriate outside layers.

HANDLING SPECIFIC ERRORS WITH EXCEPT CLAUSES

- You handle specific errors in different ways:

```
try:
    x = float(input("Enter a number: "))
    y = 100 / x
except ValueError:
    print("Bad entry.")
except ZeroDivisionError:
    print("Can't divide by zero.")
```

- How does Python execute this?
  - ➡ It executes the lines in the `try` block.
  - ➡ If no error occurs, runs as normal (skipping the code under `except`).

HANDLING SPECIFIC ERRORS WITH EXCEPT CLAUSES

- You handle specific errors in different ways:

```
try:
    x = float(input("Enter a number: "))
    y = 100 / x
except ValueError:
    print("Bad entry.")
except ZeroDivisionError:
    print("Can't divide by zero.")
```

- How does Python execute this?
  - ➡ If an error occurs within the `try` block, execution of it halts.
  - ➡ Python then jumps down to the `except` clauses.

HANDLING SPECIFIC ERRORS WITH EXCEPT CLAUSES

- You handle specific errors in different ways:

```
try:
    x = float(input("Enter a number: "))
    y = 100 / x
except ValueError:
    print("Bad entry.")
except ZeroDivisionError:
    print("Can't divide by zero.")
```

- How does Python execute this?
  - If an error occurs within the `try` block, execution of it halts.
  - Python then jumps down to the `except` clauses.
  - It executes the lines of the `except` clause that matches.

HANDLING SPECIFIC ERRORS WITH EXCEPT CLAUSES

- You handle specific errors in different ways:

```
try:
    x = float(input("Enter a number: "))
    y = 100 / x
except ValueError:
    print("Bad entry.")
except ZeroDivisionError:
    print("Can't divide by zero.")
```

- How does Python execute this?
  - If an error occurs within the `try` block, execution of it halts.
  - Python then jumps down to the `except` clauses.
  - If none match, then the error is raised.

HANDLING SPECIFIC ERRORS WITH EXCEPT CLAUSES

- You handle specific errors in different ways:

```
try:
    x = float(input("Enter a number: "))
    y = 100 / x
except (ValueError, ZeroDivisionError):
    print("Something went wrong.")
```

- How does Python execute this?
  - If an error occurs within the `try` block, execution of it halts.
  - Python then jumps down to the `except` clauses.
  - You can mention several errors to match a single clause.

HANDLING SPECIFIC ERRORS WITH EXCEPT CLAUSES

- You handle specific errors in different ways:

```
try:
    x = float(input("Enter a number: "))
    y = 100 / x
except ValueError:
    print("Bad entry.")
except ZeroDivisionError:
    print("Can't divide by zero.")
except:
    print("Something else happened.")
```

- How does Python execute this?
  - If an error occurs within the `try` block, execution of it halts.
  - Python then jumps down to the `except` clauses.
  - An `except` with no error matches every error.

TAKING CARE WITH A CATCHALL EXCEPT

- Code that catches all errors, handles them the same:

```
… stuff that computes x, y, and z …
try:
    doSomethingWith(x,y,z)
except:
    print(“Something went wrong.”)
```

- Generally it’s bad practice to hide errors with a “catchall” `try…except`.

TAKING CARE WITH A CATCHALL EXCEPT

- Code that catches all errors, handles them the same:

```
… stuff that computes x, y, and z …
try:
    doSomethingWith(x,y,z)
except:
    print(x,y,z)
    raise
```

- Generally it's bad practice to hide errors with a "catchall" `try…except`.
- Can instead print debugging information, then re-`raise` the error

TAKING CARE WITH A CATCHALL EXCEPT

- Code that catches all errors, handles them the same:

```
… stuff that computes x, y, and z …
try:
    doSomethingWith(x,y,z)
except Exception as e:
    print(“Error ‘“ + str(e) + ”’ occurred!”)
```

- Generally it’s bad practice to hide errors with a “catchall” `try…except`.
- Can instead print debugging information, then re-`raise` the error.
- Or you can at least report it and let the code keep stumbling along.
  - ➡ The code above sets `e` to the error raised.

USING EXCEPT...AS

- Example with **except…as**:

```
… stuff that computes x, y, and z …
try:
    x = float(input(“Enter a number: “)
    y = 100.0 / x
except Exception as e:
print(“Error ‘“ + str(e) + ”’ occurred!”)
```

USING EXCEPT...AS

- Example with **except…as**:

```
… stuff that computes x, y, and z …
try:
     x = float(input(“Enter a number: “)
     y = 100.0 / x
except Exception as e:
     print(“Error ‘“ + str(e) + ”’ occurred!”)
```

```
$ python3 sample_except-as_script.py
Enter a number: 0
Error 'float division by zero' occurred!
$
```

INVENTING YOUR OWN ERRORS

- You can invent and raise your own errors.

```
class BadStuffHappened(Exception):
    pass

def inputAndCompute():
    x = float(input("Enter a number: "))
    y = 100.0 / x
except (ValueError, ZeroDivisionError):
    raise BadStuffHappened("Bad stuff happened.")
```

- They can inherit from the built-in error class **`Exception`**.

GENERAL SYNTAX OF TRY… EXCEPT

- Here is general error handler code:

```
try:
    …code that might raise an error…

except SomeError1:
    …code to execute if SomeError1 occurs…

except SomeError2:
    …code to execute if SomeError2 occurs…

…

except:
    …code to execute if any other error occurs…

else:
    …code to execute if no error occurs…

finally:
    …code that runs last for all situations, error or not…
```

GENERAL SYNTAX OF TRY… EXCEPT

- Here is general error handler code:

```
try:
    …code that might raise an error…

except SomeError1:
    …code to execute if SomeError1 occurs…

except SomeError2:
    …code to execute if SomeError2 occurs…

…

except:
    …code to execute if any other error occurs…

else:
    …code to execute if no error occurs…

finally:
    …code that runs last for all situations, error or not…
```

GENERAL SYNTAX OF TRY… EXCEPT

- Here is general error handler code:

```
try:
    …code that might raise an error…

except SomeError1:
    …code to execute if SomeError1 occurs…

except SomeError2:
    …code to execute if SomeError2 occurs…

…

except:
    …code to execute if any other error occurs…

else:
    …code to execute if no error occurs…

finally:
    …code that runs last for all situations, error or not…
```

EXAMPLE USE OF FINALLY

- Used to close a file object if something happens during its use.

```
… compute x, y, z …
f = open(“to_write.txt”, “w”)
try:
    f.write(str(sqrt(x)))
    f.write(str(100 / y))
    f.write(str(f(z)))
finally:
    f.close()
```

EXAMPLE USE OF FINALLY

- Used to close a file object if something happens during its use.

```
… compute x, y, z …
f = open(“to_write.txt”, “w”)
try:
    f.write(str(sqrt(x)))
    f.write(str(100 / y))
    f.write(str(f(z)))
finally:
    f.close()
```

- This closes the file if any error occurs within the `try` block.

WITH EXAMPLE

- Can also use a `with` statement:

```
with open("to_write.txt", "w") as f:
    f.write(str(sqrt(x)))
    f.write(str(100 / y))
    f.write(str(f(z)))
```

- This also closes the file if any error occurs within the `with` block.

USING ASSERTIONS

- I've gotten into the habit of making assertions in my code.

```
…
assert(x > 0)
y = 100 // math.isqrt(x)
print("100 / sqrt(" + str(x) + ") is " + str(y) + ".")
```

- This is considered good software practice, especially as you develop code.
- They raise an **`AssertionError`**, highlighting the place where the condition failed.
- For example:

```
$ python3 sample_assert_script.py
Enter a positive integer: 0
Traceback (most recent call last):
  File "simple_assert_script.py", line 2, in <module>
    assert(x > 0)
           ^^^^^
AssertionError
$
```