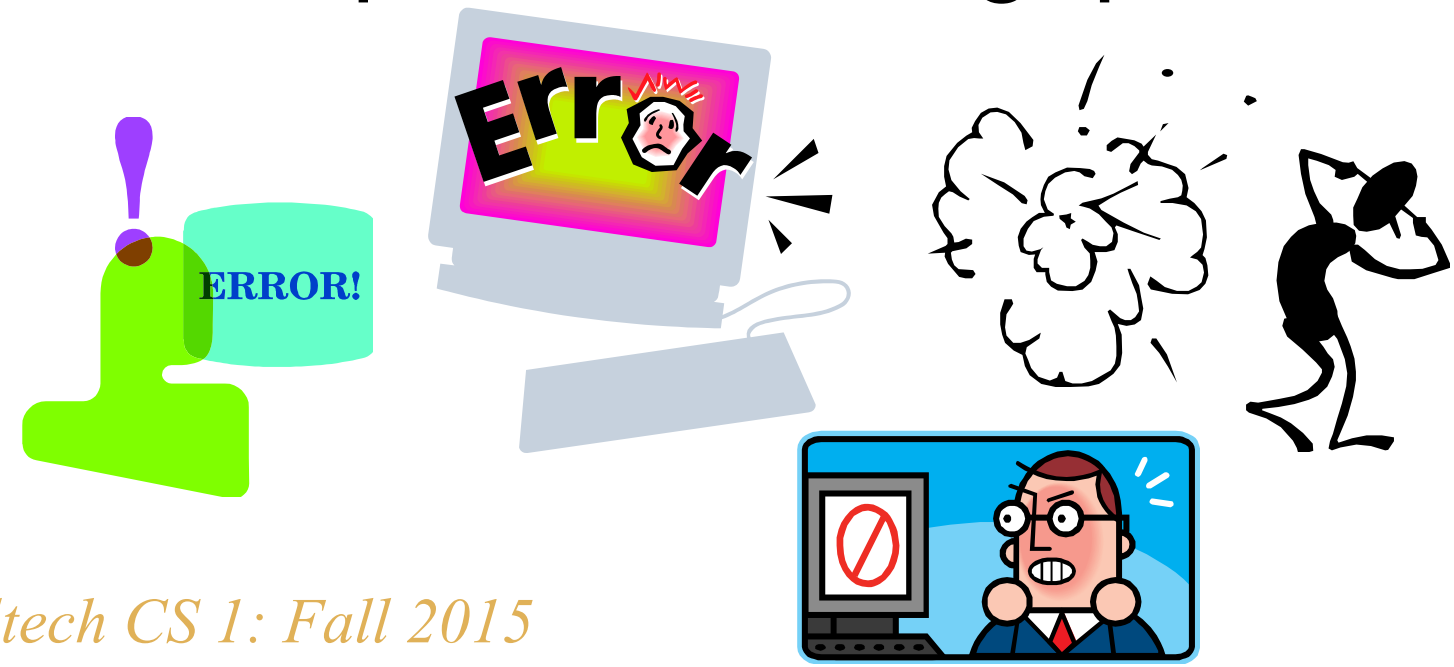
CS 1
Introduction to Computer Programming

Lecture 18: November 16, 2015

Exception Handling, part 2



Last time

- Exception handling
  - How to recover from errors
  - The `try`/`except` statement
  - Exception objects
  - Raising and catching exceptions
  - Catch-all exception handlers

Today

- More on exception handling
  - Catching exceptions outside of the function they were raised in
- The runtime stack
  - Frames
  - Tracebacks

Recall

- Last time we saw this function:

```
def sum_numbers_in_file(filename):
    sum_nums = 0
    file = open(filename, 'r')
    for line in file:
        line = line.strip()
        sum_nums += int(line)
    file.close()
    return sum_nums
```

Recall

- The purpose of `sum_numbers_in_file` is to take a filename and return a number
- The filename's file should exist and contain only numbers (one per line)
- However, errors can happen:
  - the file may not exist
  - the file may contain lines with non-numbers
  - the file may contain lines with multiple numbers
  - the file may have blank lines
- How can we handle these errors?

Error handling

- First, we need to know what kinds of Python *exceptions* are *raised* when these errors happen
- If none of the error conditions exist, the function works fine
- If the file doesn't exist, an `IOError` exception is raised when attempting to read the file
  - we saw this last time
- What about other possible errors?

Error handling

- What if the line contains something other than numbers?
- For instance, imagine this is one section of a file being read:

```
11
43
10010
thirty-five
45
```

Error handling

- Let's run this file through the function:

```
11              # OK
43              # OK
10010           # OK
thirty-five     # Error: ValueError
45
```

ValueError

- The error comes when trying to execute this line:

```
sum_nums += int(line)
```

- where `line` is `'thirty-five'`
- The `int` function can't convert the string `'thirty-five'` to an actual integer
  - it's not that smart!

ValueError

- `int('thirty-five')`
- Gives this error message:

```
ValueError: invalid literal for int()
  with base 10: 'thirty-five'
```

- Exception type: ***ValueError***
- Data associated with the exception:

```
invalid literal for int() with base 10:
  'thirty-five'
```

- gives more information about what caused the exception

ValueError

- OK, that takes care of non-numbers
- What about blank lines?
- Will end up having to compute

```
sum_nums += int(line)
```

- where `line` is `''`
  - (ending newline has been stripped off by `strip` method call)
- So the function will compute:

```
int('')
```

ValueError

```
int('')
```

- results in:

```
ValueError: invalid literal for
  int() with base 10: ''
```

- Python raises the same exception!

ValueError

- Finally: what about multiple numbers on a line?
- Will end up having to compute

```
sum_nums += int(line)
```

- where `line` is *e.g.* `'1 2 3'`
- So the function will compute:

```
int('1 2 3')
```

ValueError

```
int('1 2 3')
```

- results in:

```
ValueError: invalid literal for
  int() with base 10: '1 2 3'
```

- Python raises the same exception again!
- In all these cases, Python couldn't convert the contents of the line into an integer that could be added to the sum (`sum_nums`)

`IOError` and `ValueError`

- The function **`sum_numbers_in_file`**
  - takes in an input argument, representing a file of numbers, one per line
  - outputs the sum of all the numbers in the file
  - may raise an ***`IOError`*** if the file doesn't exist
  - may raise a ***`ValueError`*** if the file isn't formatted correctly (non-numbers or blank lines or too many numbers on some lines)
- So need to be able to handle two different kinds of error situations as well as normal case

Handling exceptions

- Last time, saw how to handle exceptions using **`try`** and **`except`**
- Put code that can raise exceptions inside a **`try`** block
- Handle the exceptions inside an **`except`** block
- Can have multiple **`except`** blocks, one per exception type to be handled
  - *e.g.* one for ***`IOError`***, one for ***`ValueError`***
- Let's try to rewrite our code accordingly

Handling exceptions

- New version, with exception handling:

```
def sum_numbers_in_file(filename):
    try:
        # Previous code
    except IOError:
        # What to do here?
    except ValueError:
        # What to do here?
```

The problem

- What code do we put in the `except` blocks?
- There are multiple ways to handle any particular error
- For an `IOError`, we could:
  - interactively prompt the user for a different file name, or
  - immediately return `0` since the file name is no good, or
  - abort entirely (don't handle the exception) and let the program terminate

The problem

- What code do we put in the `except` blocks?
- There are multiple ways to handle any particular error
- For a `ValueError`, could:
  - assume the line is no good, just use `0` as the number and keep going, or
  - assume the entire file is corrupt, return `0` from the function, or
  - abort entirely (don't handle the exception) and let the program terminate

Design decisions...

- There are a lot of choices to make!
- (9 different possible combinations of exception handling strategies in one little function!)
- And one further question:

- Should this function be in charge of deciding what happens in the event of an error?

Design decisions...

- One can imagine situations in which every possible way of handling errors would be appropriate
- If reading in a small number of files, perhaps OK to prompt user for a new filename in case the filename given doesn't exist
- If trying to read a large list of files, this would not be appropriate

Scenario 1

- Using the function interactively from the Python shell:

```
>>> sum_numbers_in_file('nums.data')
```

- If there is no file named `'nums.data'`, it's reasonable to ask the user for another filename

```
File 'nums.data' not found; please enter
  another filename: mynums.data
```

- And then the program continues, using `mynums.data` as the file to read from

Scenario 2

- Using the function from inside a larger Python program which reads a large number of files:

```
filenames = []
for i in range(1, 1001):   # 1000 files to read in
    filenames.append('day%d.data' % i)
# Now filenames is ['day1.data', 'day2.data', ...]
total = 0
for name in filenames:
    total += sum_numbers_in_file(name)
```

- Here, better to just ignore missing files (maybe also print an error message) and continue

The point

- Even though you *can* handle exceptions inside the function where they are raised, it's sometimes better *not* to
- Instead, could handle exceptions in the function which *called* the function where the exceptions might be raised
- Then, the *calling* function could decide how to handle the exceptions based on its needs

Scenario 1 again

- When running `sum_numbers_in_file` interactively, we would like to say:
- "If `sum_numbers_in_file` raises an `IOError` exception, tell the user the file doesn't exist, prompt the user for a different filename and try again"
- "If `sum_numbers_in_file` raises a `ValueError` exception, tell the user the file has invalid lines, prompt the user for a different filename and try again"

Scenario 2 again

- When running `sum_numbers_in_file` on a large number of filenames, we would like to say:
- "If `sum_numbers_in_file` raises an `IOError` exception, assume the file doesn't exist and keep going with the next file"
- "If `sum_numbers_in_file` raises a `ValueError` exception, assume the file is invalid and keep going with the next file"

"Remote" exception handling

- Python allows you to catch exceptions outside of the function in which the exception was raised
- But to understand how this works, first we have to understand the *runtime stack*
- This will involve some details about how Python works internally
  - Trust me, it's necessary knowledge!

Functions calling functions

- Consider these functions:

```
def func1(x, y):
    z = func2(x * x, y * y)
    return x + y + z

def func2(a, b):
    return (func3(a) + func3(b))

def func3(n):
    return (3 * n + 10)
```

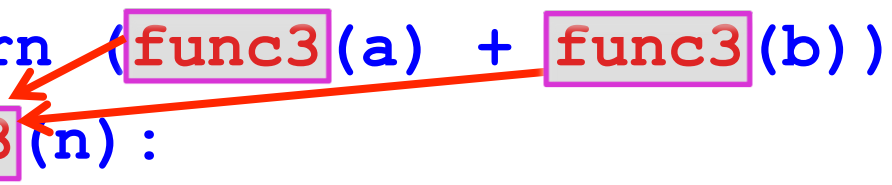
Functions calling functions

```
def func1(x, y):
    z = func2(x * x, y * y)
    return x + y + z
def func2(a, b):
    return (func3(a) + func3(b))
def func3(n):
    return (3 * n + 10)
```

- **func1** calls **func2**, which calls **func3**

Functions calling functions

```
def func1(x, y):
    z = func2(x * x, y * y)
    return x + y + z
def func2(a, b):
    return (func3(a) + func3(b))
def func3(n):
    return (3 * n + 10)
```



- **func1** calls **func2**, which calls **func3**

Function data

```
def func1(x, y):
    z = func2(x * x, y * y)
    return x + y + z
```

- Notice that when **`func1`** calls **`func2`**:
  - **`func1`** has to wait for **`func2`** to return its return value back to it before **`func1`** can complete
  - While **`func2`** is computing, the values of **`x`** and **`y`** in **`func1`** have to be stored somewhere in memory, to be used again once **`func2`** returns
- Where are **`x`** and **`y`** from **`func1`** stored while **`func2`** is running?

The runtime stack

```
def func1(x, y):
    z = func2(x * x, y * y)
    return x + y + z
```

- Answer: the values `x` and `y` from `func1` are stored in an internal Python data structure called the *runtime stack*
  - (or *"the stack"* for short)

The runtime stack

```
def func1(x, y):
    z = func2(x * x, y * y)
    return x + y + z
```

- Every time a function is called, a data structure is created which can hold all the *arguments* of the function and all *local variables*
- Here, this includes the arguments **`x`** and **`y`**, as well as the local variable **`z`**

Frames

```
def func1(x, y):
    z = func2(x * x, y * y)
    return x + y + z
```

- This data structure is called a *frame*
- It's kind of like a dictionary
  - mapping names (like `x`, `y`, `z`)
  - to their current values in *this particular call of the function*

Frames

```
def func1(x, y):
    z = func2(x * x, y * y)
    return x + y + z
```

- Frames are *temporary* data structures
- They are created when a function is called
- They go away when the function returns
- They are used by Python to look up the current value of any local variable or function argument while evaluating the body of a function

Frames

```
def func1(x, y):
    z = func2(x * x, y * y)
    return x + y + z
```

- If we called **func1** like this:

```
>>> func1(10, 20)
```

- Then the frame for this function call would have:

```
x → 10
y → 20
```

Frames

```
def func1(x, y):
    z = func2(x * x, y * y)
    return x + y + z
```

- What happens when one function (like **`func1`**) calls another function (like **`func2`**)?
- The first function hasn't returned yet, so its frame still exists
- The second function gets a frame too, as soon as it gets called

Frames

```
def func1(x, y):
    z = func2(x * x, y * y)
    return x + y + z
```

- Answer: the frames from the different functions are kept on a *stack* of frames
- Every time a function is called, a new frame is created and *pushed* onto the stack
- Every time a function returns, its frame is *popped* off the stack

Frames

```
def func1(x, y):
    z = func2(x * x, y * y)
    return x + y + z
```

- Frames also keep track of the exact location in the code where a function (like **`func1`**) called another function (like **`func2`**)
  - so when **`func2`** returns, Python knows where to continue in **`func1`**

Frames

```
def func1(x, y):
    z = func2(x * x, y * y)
    return x + y + z
```

- Frames also keep track of the exact location in the code where a function (like **`func1`**) called another function (like **`func2`**)
  - so when **`func2`** returns, Python knows where to continue in **`func1`**
  - (assign the return value of **`func2`** to **`z`** and continue)

Functions calling functions

- Recall:

```
def func3(n):
    return (3 * n + 10)
```

- When we call this function:

```
>>> func3(10)
```

- Then a frame is created which has only

`n → 10`

- It computes its return value (`40`) and then returns to its caller
  - and the frame goes away (not needed anymore)

Functions calling functions

- A more complicated example:

```
def func2(a, b):
    return (func3(a) + func3(b))
def func3(n):
    return (3 * n + 10)
```

- Let's do this:

```
>>> func2(10, 20)
```

Functions calling functions

```
def func2(a, b):
    return (func3(a) + func3(b))

def func3(n):
    return (3 * n + 10)
```

- `func2(10, 20)` starts executing
- This creates a frame with `a` bound to `10` and `b` bound to `20`
- This frame is pushed onto the stack

`func2`

`a` → 10
`b` → 20

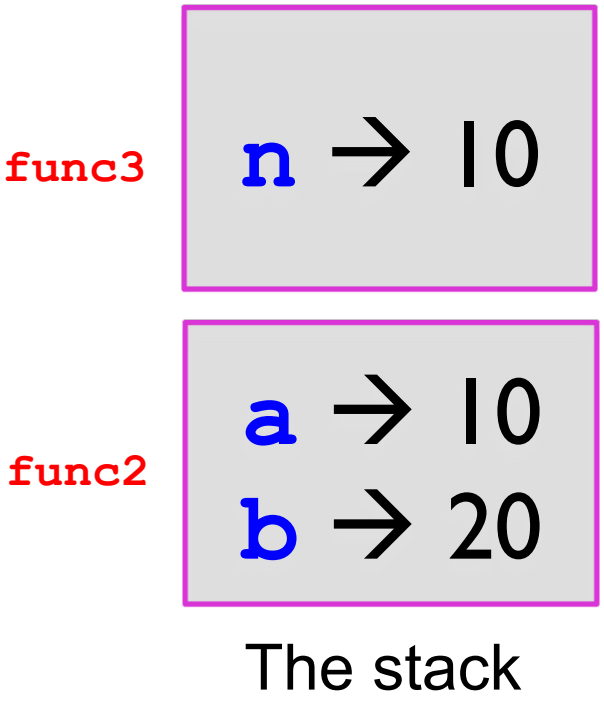
The stack

Functions calling functions

```
def func2(a, b):
    return (func3(a) + func3(b))

def func3(n):
    return (3 * n + 10)
```

- **func2** calls **func3(a)**
- which is **func3(10)**
- This causes **func3** to create a new frame with **n** bound to **10**
- This frame gets pushed onto the stack above **func2**'s frame



The stack

Functions calling functions

```
def func2(a, b):
    return (40 + func3(b))

def func3(n):
    return (3 * n + 10)
```

- `func3(10)` evaluates to `40`
- This value is returned to `func2`, and the frame for `func3` is removed ("popped") from the stack

`func2`

a → 10
b → 20

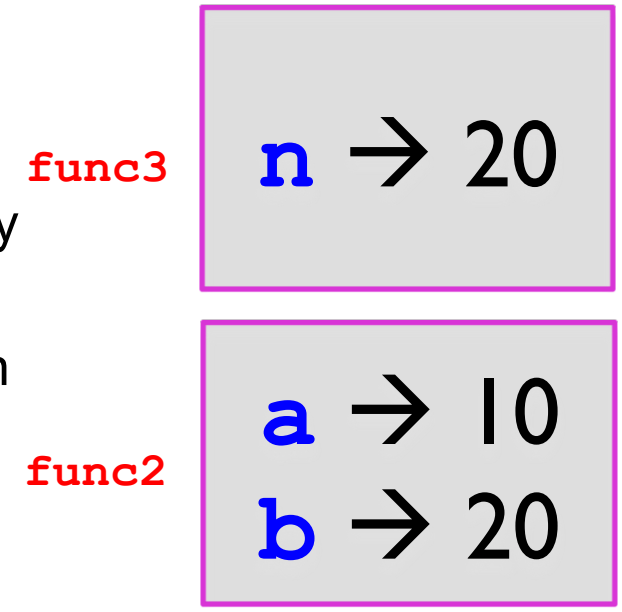
The stack

Functions calling functions

```
def func2(a, b):
    return (40 + func3(b))

def func3(n):
    return (3 * n + 10)
```

- Now `func3(b)` is called
- `b` is `20`, so `func3(20)` is what's really called
- A new frame for `func3` is created with `n` bound to `20`, and pushed onto the stack
- Then the body of `func3` is evaluated



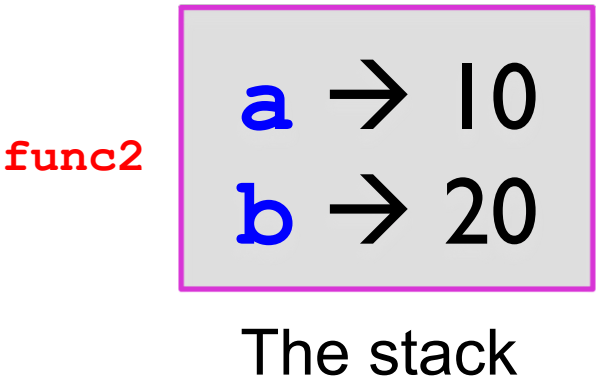
The stack

Functions calling functions

```
def func2(a, b):
    return (40 + 70)

def func3(n):
    return (3 * n + 10)
```

- `func3(20)` evaluates to `70`
- This value is returned to `func2`, and the frame for `func3` is removed ("popped") from the stack



The stack

Functions calling functions

```
def func2(a, b):
    return (110)

def func3(n):
    return (3 * n + 10)
```

- `func2(10, 20)` returns `110`
- The frame for `func2` is removed ("popped") from the stack
- Now the stack is empty

The stack

Functions calling functions

- This is how function calls work in Python
  - (and in almost every computer language!)
- The top of the stack contains the frame for the currently executing function
- When one function calls another function, which calls another function, etc.
  - the stack grows, with one stack frame per function call
- As the function calls complete their work and return
  - stack frames get removed from the stack

What about exceptions?

- How does any of this relate to exception handling?
- We'll see after the break...

Interlude

- It's *insanely great!*

Recall

- We have a function called `sum_numbers_in_file`
- It takes a filename, returns a number
- Can raise an `IOError` exception if the file with the filename doesn't exist
- Can raise a `ValueError` exception if the file has lines without numbers
- Don't want to handle these exceptions in the function itself
  - too many different ways to do it

"Remote" exception handling

- We *won't* have a `try` block inside the `sum_numbers_in_file` function
- We *will* have a `try` block inside the function that *calls* the `sum_numbers_in_file` function
  - (actually, doesn't have to be inside a function)
- Let's write two different functions that call `sum_numbers_in_file` and handle the possible exceptions

Function #1

- This function will try to run `sum_numbers_in_file` on a single filename
- If an exception happens, it will prompt for a new filename and will try again
- If no exception happens, it will return the sum of the numbers in the file

Function #1

```
def sum_numbers_in_file_interactive(filename):
    while True:
        try:
            total = sum_numbers_in_file(filename)
            return total
        except IOError:
            print 'Couldn't read file: %s' % filename
            filename = raw_input('Enter new filename: ')
        except ValueError:
            print 'Invalid line in file: %s' % filename
            filename = raw_input("Enter new filename: ')
```

Function #1

- In this function, **`sum_numbers_in_file`** is called inside a **`try`** block
- If an exception is raised during the execution of **`sum_numbers_in_file`**, it will not be caught in that function
- Instead, the exception will go back down in the runtime stack to the function that called it (**`sum_numbers_in_file_interactive`**)
- And it will get handled inside the **`try`**/**`except`** block of this function

Function #1

- This function will continue to run `sum_numbers_on_file` on every filename supplied by the user until a valid filename is given
  - *i.e.* a filename for which a sum can be computed
- Once a sum is computed, the function will return
  - which breaks us out of the `while True:` loop

Exceptions and the stack

- The runtime stack is important when dealing with exceptions that are not caught inside the function in which they were raised
- These exceptions "propagate back" in the stack to the function that *called* the function where the exception was raised
  - *i.e.* they pop the stack and go down one stack frame (like a special kind of `return` but without a return value)
- If the exception *can* be handled inside a `try`/`except` statement in the calling function, it will be

Exceptions and the stack

- If the exception is *not* handled in the calling function, it will propagate back one *more* stack frame, looking for a `try`/`except` statement that can catch it
- It will keep doing this until either
  1. it finds a suitable `except` block, in which case it will execute the code in that block, or
  2. it reaches the top level of the program (no more stack frames), in which case the program will halt

Exceptions and the stack

- What is a *"suitable* **`except`** *block"*?
- It's an except block labeled with the same name as the exception that was raised
- So if a **`ValueError`** is raised, a suitable except block is:

```
except ValueError:
    # … contents of block ...
```

Unwinding the stack

- This process, by which exceptions propagate back down the stack, looking for a suitable `except` block, is called *unwinding the stack*
- Once the exception has left a function's frame, that frame is popped off the stack, and that function call is over (just as if the function had returned)
- Exceptions continue to unwind the stack until they are caught (by a suitable `except` block) or until they reach the top level of the program

Unwinding the stack

- If the exception goes all the way to the top level (no more stack frames to pop), then it halts the program and prints a *traceback*
- A traceback is a record of exactly where the program was when the exception was raised
  - including the position in the file of every function call in the stack
- A traceback is useful information!
  - Can show you where bugs are

Traceback example

- Consider a Python module called **dumb.py**:

```
1 def func1(n):
2     return (2 * func2(n))
3 def func2(n):
4     return (3 * func3(n))
5 def func3(n):
6     return (1 / 0)
7 if __name__ == '__main__':
8     print func1(42)
```

- What will happen when we run this file?

Traceback example

```
% python dumb.py
Traceback (most recent call last):
  File "dumb.py", line 8, in <module>
    print func1(42)
  File "dumb.py", line 2, in func1
    return (2 * func2(n))
  File "dumb.py", line 4, in func2
    return (3 * func3(n))
  File "dumb.py", line 6, in func3
    return (1 / 0)
ZeroDivisionError: integer division or modulo by
  zero
```

Traceback example

```
% python dumb.py
Traceback (most recent call last):
  File "dumb.py", line 8, in <module>
    print func1(42)
  File "dumb.py", line 2, in func1
    return (2 * func2(n))
  File "dumb.py", line 4, in func2
    return (3 * func3(n))
  File "dumb.py", line 6, in func3
    return (1 / 0)
```

- This line was executed first

Traceback example

```
% python dumb.py
Traceback (most recent call last):
  File "dumb.py", line 8, in <module>
    print func1(42)
  File "dumb.py", line 2, in func1
    return (2 * func2(n))
  File "dumb.py", line 4, in func2
    return (3 * func3(n))
  File "dumb.py", line 6, in func3
    return (1 / 0)
```

- Which called this function

Traceback example

```
% python dumb.py
Traceback (most recent call last):
  File "dumb.py", line 8, in <module>
    print func1(42)
  File "dumb.py", line 2, in func1
    return (2 * func2(n))
  File "dumb.py", line 4, in func2
    return (3 * func3(n))
  File "dumb.py", line 6, in func3
    return (1 / 0)
```

- Which called this function

Traceback example

```
% python dumb.py
Traceback (most recent call last):
  File "dumb.py", line 8, in <module>
    print func1(42)
  File "dumb.py", line 2, in func1
    return (2 * func2(n))
  File "dumb.py", line 4, in func2
    return (3 * func3(n))
  File "dumb.py", line 6, in func3
    return (1 / 0)
```

- Which called this function

Traceback example

```
% python dumb.py
Traceback (most recent call last):
  File "dumb.py", line 8, in <module>
    print func1(42)
  File "dumb.py", line 2, in func1
    return (2 * func2(n))
  File "dumb.py", line 4, in func2
    return (3 * func3(n))
  File "dumb.py", line 6, in func3
    return (1 / 0)
```

- Which is where the error occurred

Traceback example

- At the end of the traceback, the exception type and error message (if any) is printed
- A traceback is *only* printed if the exception is not caught by a suitable `try`/`except` statement

Back to our function

- We showed how to write a function which calls `sum_numbers_in_file` and asks for a new filename when exceptions are raised
- If we were dealing with a large number of files, this wouldn't be practical
- Let's write a function to deal with this case
- When an exception occurs:
  - an error message will be printed
  - but execution will continue

Function #2

```
def sum_numbers_in_multiple_files(filenames):
    total = 0
    for filename in filenames:
        try:
            total += sum_numbers_in_file(filename)
        except IOError:
            print 'Couldn't read file: %s' % filename
        except ValueError:
            print 'Invalid line in file: %s' % filename
    return total
```

Function #2

- When this function executes, exceptions in `sum_numbers_in_file` due to bad or missing files simply cause those files to be ignored
- More appropriate for a program which normally runs non-interactively
- In real program, error messages would probably be written out to a log file

Question

- How to modify the second example in the case where:
  - missing files are skipped over with an error message (same as previous example)
  - bad lines are skipped over with an error message but the file as a whole is *not* skipped over (the sum of the numbers from properly formatted lines in the file is what's returned)
- Hint: Need `try`/`except` statements in both `sum_numbers_in_file` and `sum_numbers_in_multiple_files`

Next time

- More object-oriented programming
  - class "inheritance"
- Exceptions and classes
  - creating your own exceptions
  - raising exceptions explicitly
  - the `Exception` base class

Preview: This Friday

- First "CS 2 prep" lecture
- Will introduce a brand-new computer language: C++
  - Very different from Python!
- We'll emphasize "survival skills" to get you ready for CS 2

Preview: Next Wednesday

- Special topic lecture: regular expressions
  - Extremely powerful string processing concepts
  - Very heavily used in practice (bioinformatics, web programming, elsewhere)