ENGR 102 PROGRAMMING PRACTICE

WEEK 2

Databases (File-based Dictionaries)

- A database is a file that is organized for storing data.
- Most databases are organized like a dictionary
  - Keys → Values
  - The biggest difference:
    - a database is on disk (or other permanent storage), so it persists after the program ends.
- The module **anydbm** provides an interface for creating and updating database files.

Example

- Write a program that saves, in a db, letter and attendance grades of students.
  - Write a function that returns the average attendance grade of the students in the db.

Databases (File-based Dictionaries)

- Opening a database is similar to opening files:

```
import anydbm

db = anydbm.open('captions', 'c')
```

- Mode 'c':
  - database should be created if it doesn’t already exist.
- Returns a database object that can be used (for most operations) like a dictionary.

Databases (File-based Dictionaries)

- If you create a new item, anydbm updates the database file.

```
db['cleese.png'] = 'Photo of John Cleese.'
```

- When you access one of the items, anydbm reads the file:

```
print db['cleese.png']
Photo of John Cleese.
```

Databases (File-based Dictionaries)

- If you make another assignment to an existing key, anydbm replaces the old value in the file:

```
db['cleese.png'] = 'Photo of John Cleese eating.'
print db['cleese.png']
Photo of John Cleese eating.
```

Databases (File-based Dictionaries)

- Some dictionary methods, like keys(), values() and items(), also work with database objects. You may iterate over keys with a for statement.

```
for key in db.keys():
    print key
```

- As with other files, you should close the database when you are done:

```
db.close()
```

Warning:
If you are using Apple-originated 2.7 interpreter db.values() and db.items() may not work.
Use db.keys() if this is the case.

Databases
Mode Flags

Value	Meaning
'r'	Open existing database for reading only (default)
'w'	Open existing database for reading and writing
'c'	Open database for reading and writing, creating it if it doesn’t exist
'n'	Always create a new, empty database, open for reading and writing

Pickling

- A **limitation** of anydbm is that the **keys and values** have to be **strings**.
  - If you try to use any other type, you get an error.

- **pickle** module may help.
  - Object → String [pickle.dumps(object)]
  - String → Object [pickle.loads(str)]

Pickling

- **pickle.dumps** takes an object as a parameter and returns a string representation
  - (dumps is short for "dump string").

```
import pickle
t1 = [1, 2, 3]
print pickle.dumps(t1)
(lp0\nI1\naI2\naI3\na.
```

Pickling

- Although the new object has the same value as the old, it is not (in general) the same object:

```
str = pickle.dumps(t1)

t2 = pickle.loads(str)

print t1 == t2
True

print t1 is t2
False
```

Exceptions

Simple addition

- Write a program that prompts the user for an integer n and prints n+1
  - keep asking until the user enters a valid input (that can be converted to an integer)

Factorial

- Write a function that takes an integers n
  - if n>=0, returns n!
  - if n<0, raise ValueError
  - in case of TypeError, print an error message

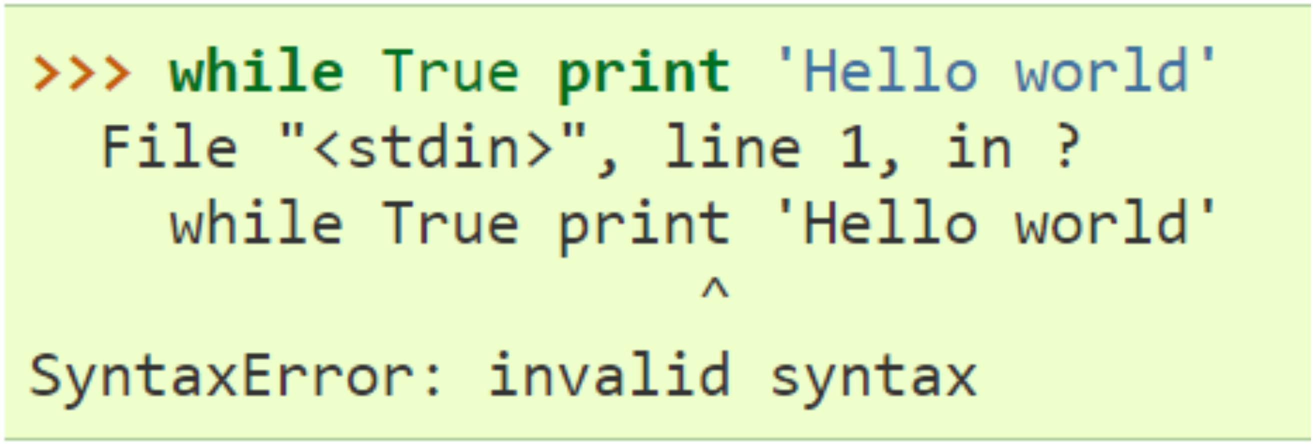
- Call the function in main, handle ValueError exception

The world is not perfect!

```
fin = open('bad_file.txt')
for line in fin:
    print line
fin.close()
```

Exceptions

It's all about errors. What kind?



```
>>> while True print 'Hello world'
  File "<stdin>", line 1, in ?
    while True print 'Hello world'
                   ^
SyntaxError: invalid syntax
```

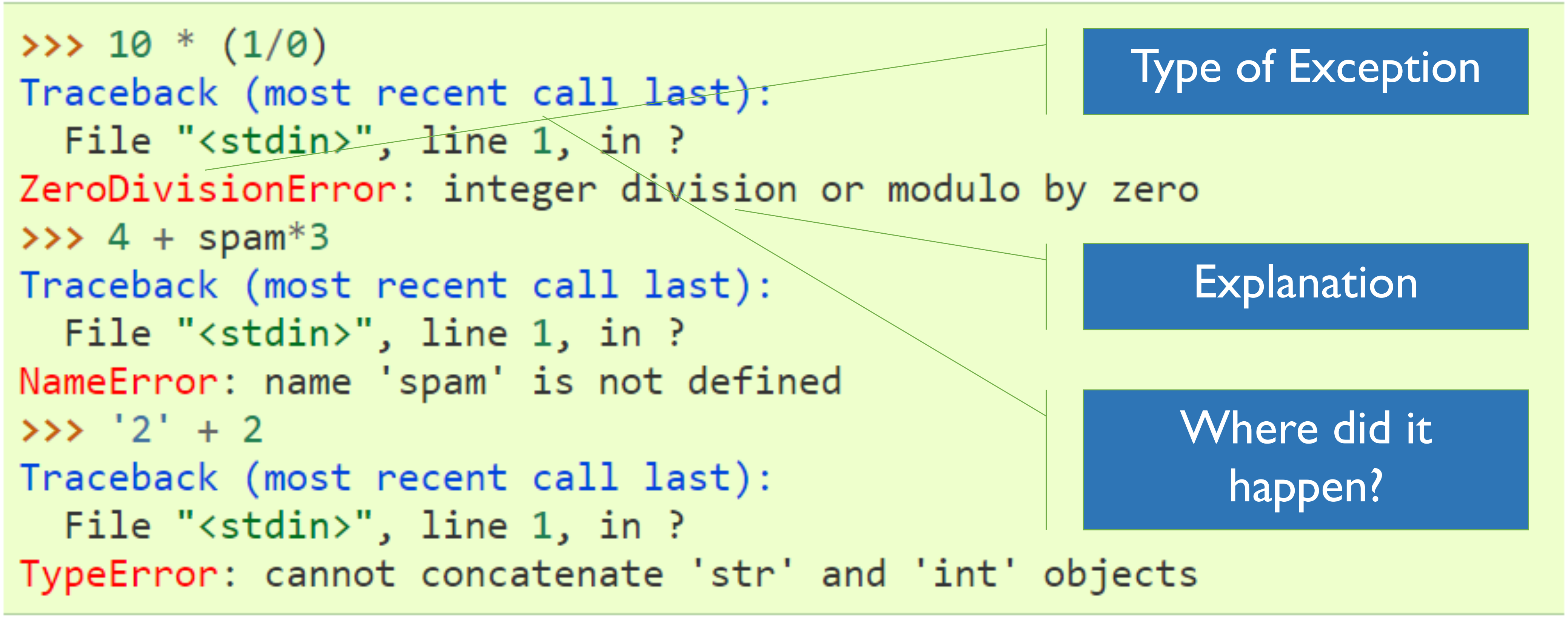
These are parse-time errors

→ detected before running your program.

Exceptions are errors detected during execution!

How do you handle Exceptions?

- Even before that:
  - What happens if you do not handle them?



```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

Catching exceptions

- try - except clause

```
while True:
    try:
        x = int(raw_input("Please enter a number: "))
        break
    except ValueError:
        print "Oops!  That was no valid number.  Try again..."
```

- except clause may have multiple exception types

```
except (RuntimeError, TypeError, NameError):
    pass
```

Multiple exception types

Catching exceptions

- multiple except clauses

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as e:
    print "I/O error({0}): {1}".format(e.errno, e.strerror)
except ValueError:
    print "Could not convert data to an integer."
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

Catching exceptions

- optional else clause

```
for filename in filenames:
    try:
        f = open(filename, 'r')
    except IOError:
        print 'cannot open', filename
    else:
        print filename, 'has', len(f.readlines()), 'lines'
        f.close()
```

- else block is executed if no exception is thrown.

Printing exception details

- use as clause to get and print the exception object

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as detail:
...     print 'Handling run-time error:', detail
...
Handling run-time error: integer division or modulo by zero
```

- This is possible because __str__ method is implemented in Exception class.

Raising exceptions

- use raise clause to throw an exception

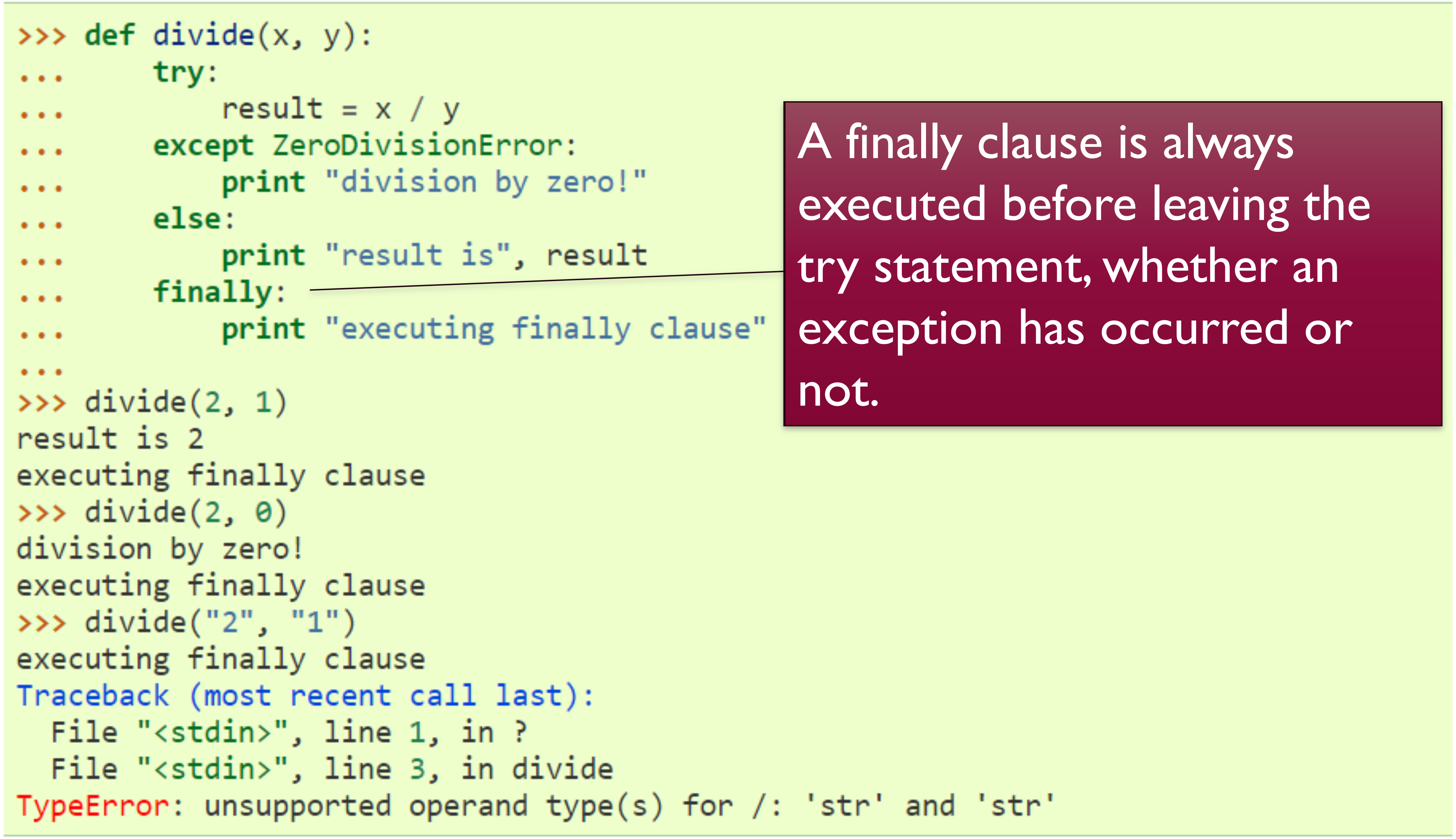
```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: HiThere
```

Raising exceptions

- If you don't intend to handle an exception, the raise statement with no input allows you to re-raise the exception

```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print 'An exception flew by!'
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: HiThere
```

Defining clean-up actions



```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print "division by zero!"
...     else:
...         print "result is", result
...     finally:
...         print "executing finally clause"
...
>>> divide(2, 1)
result is 2
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

A finally clause is always executed before leaving the try statement, whether an exception has occurred or not.