

CS 115 Lecture 17

Exceptions and files

Neil Moore

Department of Computer Science
University of Kentucky
Lexington, Kentucky 40506
`neil@cs.uky.edu`

17 November 2015

Run-time errors

Remember the three kinds of errors:

- Syntax error (can't even run the code).
- Run-time error (detected when the code runs; crashes).
- Semantic error (not detected: program does the wrong thing).

Sometimes you might want to signal a run-time error yourself. Why?

- If you encounter a situation you can't handle.
 - ▶ Usually better to handle it with an `if!`
 - ▶ But sometimes that's not possible: `float(input(...))`
- If your function's preconditions are violated.
- A run-time error is better than a semantic error.
 - ▶ At least you know it's an error!

Exceptions

Another name for a run-time error in Python is an **exception**.

- “Exception, not the rule”
- Signalling a run-time error is caused **raising** an exception.
 - ▶ Also called “throwing” an exception (C++ and Java)
 - ▶ Python does this automatically in several situations.
- By default, raising an exception crashes your program.
 - ▶ But exceptions can be **caught** and handled.
- Many different kinds of exceptions:
 - ▶ `TypeError`: argument has the wrong type.
 - ▶ `ValueError`: argument has a good type but a bad value.
 - ▶ `IndexError`: accessing a sequence out of range.
 - ▶ `ZeroDivisionError`: exactly what it says.
 - ▶ `IOError`: file problem, such as “file not found”.
 - ▶ `RuntimeError`: “none of the above”.
- <https://docs.python.org/3/library/exceptions.html>

Raising exceptions

- To raise an exception, use the raise keyword.
- You have to say which kind of exception:

```
raise ValueError("Empty list provided to minimum.")
raise ZeroDivisionError()
```
- The various kinds of exceptions are all classes.
 - ▶ Call the constructor with an optional message (a string).
- The exception name and string will appear in the crash message:

```
Traceback (most recent call last):
```

```
...
```

```
ValueError: Empty list provided to minimum.
```

Catching exceptions

By default, exceptions cause the program to crash.

- Because that's better than continuing and doing the wrong thing.
- But sometimes you might have a better idea.
- For example, type-casting a string to `int`.
 - ▶ If the string wasn't numeric, Python can't give you a number.
 - ▶ You asked Python to do something and it can't.
 - ★ Exception!
 - ▶ But maybe you know what to do in this particular case.
 - ★ If it was user input, repeat the loop and ask again.
 - ★ If it came from a file, maybe ignore that line.
- This is especially important when you can't check in advance whether an exception is going to be raised.
 - ▶ We'll see this later with `IOError`.

try/except

To catch an exception, you use a **try/except** statement:

try:

body that might raise an exception

except *ExceptionClass:*

handle the exception

following code

- *ExceptionClass* is one of `ValueError`, `IOError`, etc...
- If the body raises the specified exception:
 - ▶ The body stops executing immediately (like a “go to”).
 - ★ Doesn't even finish the current line.
 - ▶ Then Python runs the `except` block (instead of crashing).
 - ▶ After the body or the handler, go on to following code.
- This applies even if the exception is raised inside a function call!
 - ▶ Exceptions go up the call stack looking for a handler.
- Can have several `except` blocks for different exceptions.
 - ▶ `trysqrt.py`
 - ▶ Or one block for several exceptions:
`except (ValueError, IndexError): # need parentheses!`

An exception example

Let's go back to the numeric input example.

- Suppose we want to keep asking for a float until we get one.
 - ▶ So this will be an input validation loop (sentinel logic).
 - ▶ We'll use a flag to mark whether we got a good input.

```
while not ok:
```

- How do we get the input and convert to a number?

```
number = float(input("Please enter a number: "))
```

- ▶ `float(...)` raises a `ValueError` on non-numeric input.
 - ▶ So put the line inside a `try`.
 - ▶ If we catch the exception, set the `ok` flag to `False`.
 - ▶ If there wasn't an exception, set the flag to `True`. Where?
 - ★ In the `try` body after the input.
- Finally, put that whole `try/except` in two places:
 - ▶ Before the loop, and as the last step of the loop.
 - ▶ When the loop finishes, we know we have a number.

- `nonnumeric.py`

Another way

Another way to do loops involving an exception.

- Use a flag like we did before, initialized to False.
- Set the flag to True in the `try` as before.
- Put the input `try/except` *inside* the loop only.
 - ▶ Because the flag is False, the loop will run at least once.
- Put the error message in the `except`.
 - ▶ (So it only happens if there was an exception.)
- `nonnumeric2.py`
- We'll see this again when re-prompting for a file.

Hints for catching exceptions

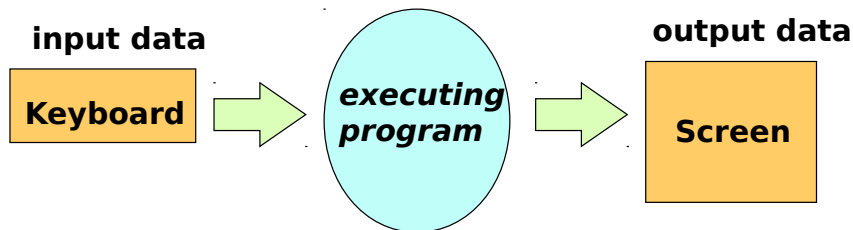
- “Do not summon up that which you cannot put down.”
—H.P. Lovecraft, “The Case of Charles Dexter Ward”
 - ▶ Have a plan!
 - ▶ If you don't know how to fix the error, don't catch it.
 - ▶ It's better to crash than to continue with bad data.
- Keep the `try` block as small as possible.
 - ▶ It should contain the line that might raise the exception.
 - ▶ And subsequent lines that depend on its success.
 - ▶ Don't duplicate code in the `try` and `except`.
 - ★ That code should come after the `try/except` so it happens either way.
 - ▶ Don't wrap the whole `main` in a `try`!
 - ★ `main` probably doesn't know how to fix the error.
- If you can use it, **if** is usually simpler.
 - ▶ If you know in advance what situations will cause an error.

Dealing with lots of data

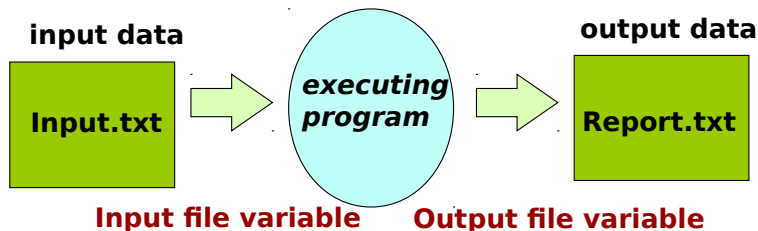
Some program need a lot of data. What to do?

- Hard code it (write it in your source)?
That's hard for non-programmers to change.
- Ask the user to type it in each time?
If it's a lot of data, your users will hate you.
- Do you have to type your source code every time you run it?
No—you save it in a **file**.
- Why use files?
 - ▶ Easier to edit than source.
 - ★ Especially if you want to change it during a run.
 - ▶ Files persist across runs of your program.
 - ★ And across reboots of your operating system.
 - ★ Can save output for later use.
 - ▶ Can hold large amounts of data (more than fits in RAM).
 - ▶ Can use the same data as input to different programs.

Input/output with the user



I/O with files



Using files

As in other programs (word processors, IDEs, etc.), you must **open** a file before you can use it in your program.

- Create a **file object** in your program that represents the file on disk.
 - ▶ You can read from and/or write to the object.
 - ▶ Input-output from/to the file instead of the user.
- Syntax:

```
fileobj = open(filename, "r") # r for reading
fileobj = open(filename) # default is reading
```

- ▶ fileobj is a variable that will hold the file object
- ▶ filename is a string that names the file.
 - ★ By default, looks for that file in the current directory.
 - ★ You can specify an absolute path instead:

```
open("C:\\Users\\me\\input.txt")
```
 - ★ Don't do this in your 115 programs: your TA probably uses different directories.
- Can also open a file for writing:

```
fileobj = open(filename, "w") # w for write
```

IOError

If we are trying to read from a file, what can go wrong?

- Maybe the file isn't there.
- Or it's there, but you don't have permissions to access it.
- Or you do, but then your hard drive crashes.
- In these situations, opening a file raises a `IOError` exception.
 - ▶ Renamed to `OSError` in Python 3.4.
- You can catch the exception just like any other.
 - ▶ But there's no point in trying again with the same filename.
 - ▶ Maybe ask the user for a new filename.

```
ok = False
while not ok:
    try:
        fn = input("Enter a file name: ")
        infile = open(fn, "r")
        ok = True
    except IOError:
        print("Could not open", fn)
```

Looping over the lines in a file

The simplest way to use an input file once you have opened it:

- Loop over the lines of the file.
- A file object can be used as a sequence of lines:
`for line in file:`
 - ▶ Each line is a string.
 - ▶ `file` should be a file object, *not* a filename.
- Beware: the line ends in a newline character!
 - ▶ You might need to use `strip` or `rstrip`.
- When you're finished with the file, **close** it:
`file.close()`
 - ▶ Frees up resources associated with the file.
 - ▶ If you don't, the file will take up memory until the program exits.
 - ▶ More on this later.
- `readfile-for.py`

Text files: characters and bytes

Files are stored on disk as a sequence of **bytes**.

- A byte is a collection of eight bits (ones or zeros)
 - ▶ Can represent a number from 0 to 255.
- In **text files**, bytes are used to encode characters.
- An **encoding** says how to translate between bytes and characters.
 - ▶ ASCII: one byte, one character—more than enough for English.
 - ▶ Latin-1, KOI8-R, . . . : Use the leftover numbers for more characters.
 - ★ But 256 characters is not enough for CJK (Chinese, Japanese, and Korean).
 - ▶ Unicode: more than 256 different characters.
 - ★ So you need multiple bytes per character.
 - ★ UTF-8, UCS-4, UTF-16: different encodings of Unicode.
 - ★ UTF-8 is ASCII-compatible, so is the most commonly used.
 - ★ (It's the default for text files in Python).
- **Text file**: stores a sequence of **characters**.
- **Binary file**: stores a sequence of **bytes**.

Text files: lines

So if a text file is just a sequence of characters, what is a line?

- A sequence of characters. . .
- There's one character that can't appear in the middle of a line.
 - ▶ The newline character!
 - ▶ Newline ('`\n`') is the **line delimiter**.
 - ★ (Technically it's a little more complex on Windows, but Python mostly hides that complexity.)
- What would two newlines in a row mean?
 - ▶ There's an empty line between them.
- So this file:

```
Hello, world.
```

```
How's it going?
```

would look like:

```
Hello, world.\n\nHow's it going?\n
```

Sequential and random access

- **Sequential access:** reading (or writing) the file in order starting from the beginning.
 - ▶ Like a for loop.
 - ▶ Read the first line first, then the second line, etc.
- **Random access:** reading or writing out of order.
 - ▶ “Go to byte 7563 and put a 1 there.”
 - ▶ Like lists: we can say `mylist[5]` without having to go through indices 0 through 4 first.
- Random access doesn't work that well with text files.
 - ▶ Bytes don't always match up with characters.
 - ▶ And they definitely don't match up well with lines...
At what byte number does line 10 start?
 - ★ You'd have to go through the lines sequentially and count!
- **Text files:** usually **sequential access**.
- **Binary files:** either sequential or **random access**.