ECE326

PROGRAMMING LANGUAGES

Lecture 5b : Files and Exceptions

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2020

Files

- Use open built-in function

```
>> f = open(“hello.txt”) # read-only mode, file must exist
>> h = open(“io.h”, “w”) # write-only mode, file will be wiped
```

- File access mode
  - Same as fopen() in C
  - Default is “r”

Files

- Reading text files

```
>> for line in f:                  # file objects are iterable
..     print(line)
# --- or ---
>> f.read()                        # read the entire file
```

- Close file (after finished)

```
>> f.close()
```

- Writing text files

```
>> h.write(“hello world\n”)    # add a new line to sentence
```

Error Handling

- Deals with runtime error without crashing
  - Type of error can range from small to critical
    - E.g. KeyError vs. ZeroDivisionError
- Separate error handling code from normal code
  - Improves readability
- Need to disrupt normal execution flow
  - Switch over to error handling code

Example

- goto statement in C
  - Execution jumps to label
- Problem
  - Can only handle exception within the same function
  - Difficult to pass information to error handling code

```
    int i;
    Dir * d = malloc(sizeof(Dir)*NUM_DIRS);
    if (d == NULL) goto fail;
    for (i = 0; i < NUM_DIRS; i++) {
        if (!(d[i] = alloc_dir()))
            goto fail2;
    }
    /* do stuff with d */
    return 0;
fail2:
    for (i-- ; i >= 0; i--)
        free_dir(d[i]);
    free(d);
fail:
    return -ENOMEM;
```

Error Handling

- C++/Python: `try` statement
- Jumps to exception handler on error
  - May need to unwind stack frames (function calls)
  - Can be expensive (C++ compile option `--fno-exceptions`)

```
try:
    f = open(“hello.txt”)
except OSError as err:
    print(err)
else:                            # if no error occurs
    print(f.read())              # this reads everything
    f.close()
```

With Statement

- Some objects have pre-defined clean-up actions
  - Special `__exit__` method
- Makes code look much cleaner

```
# close called automatically when exiting block

with open(“hello.txt”) as f:
    print(f.read())

# Note: f still in scope here (but is closed)
```

User-Defined Exception

- Create a class derived from base Exception class

```
class MyError(Exception):
    pass

>> raise MyError(“It’s bad”)       # raise your own exception
__main__.MyError: It's bad

pr = analyze_move(mv)
if pr > 1.0:                       # use built-in exception
    raise ValueError(“probability can’t be > 1!”)
```

Multiple Exceptions

```
def baz():
      try:
            foo()       # exceptions can be raised from inside
            bar()       # a function call for caller to handle
            …
      except (KeyError, ValueError):
            # deal with these two the same way
            return 0
      except OSError as err:
            print(err)
            return -1
      except:
            print(“unexpected exception!”)
            raise       # re-raise the exception to
      …                 # caller of this function
```

Scope and Except

- An “exception” to function scope
  - Exception variable is deleted at end of block

```
def try_fail():
    e = "hello"
    try:
        a = range(2)
        print(a[3])         # this will cause IndexError
    except IndexError as e:
        print(e)

    # NameError: name 'e' is not defined
    print(e + " world")
```

Scope and Except

- Why the “exception”?
  - Exception variable interferes with garbage collection
  - Potentially large amount of memory cannot be reclaimed until exception variable is deleted

- Workaround
  - If you want to keep it, reassign it to a different name

```
keep_me = None
try:
    …
except IndexError as e:
    keep_me = e

# this works
print(keep_me)
```