Data Structures in Python

5. Exception Handling (II)

Prof. Moheb Ramzy Girgis
Department of Computer Science
Faculty of Science
Minia University

Raising Exceptions

- We have seen how to write code that reacts to exceptions.
- We know that certain functions, like open and int can raise exceptions. Also, statements that attempt to divide by zero or print an undefined variable will raise exceptions.
- The following statement raises a ***ValueError*** exception:
  x = int('x')
- The following statement is a more direct way to raise a ***ValueError*** exception:
  raise ValueError()
- The ***raise*** keyword raises an exception. Its argument, if present, must be an ***exception object***.
- The *class constructor* for most exception objects accepts a string parameter that provides additional information to handlers.

Raising Exceptions

- **Example:** In the interactive shell:

```
>>> raise ValueError()
Traceback (most recent call last):
  Python Shell, prompt 4, line 1
builtins.ValueError
>>> raise ValueError('This is a value error')
Traceback (most recent call last):
  Python Shell, prompt 5, line 1
builtins.ValueError: This is a value error
```

- This means we can write a custom version of the ***int*** function that behaves similar to the ***built-in int*** function, as the following code illustrates.

```
def non_neg_int(n):
    result = int(n)
    if result < 0:
            raise ValueError(result)
    return result
```


Raising Exceptions

```
while True:
    x = non_neg_int(input('Please enter a nonnegative integer:'))
    if x == 999: # Secret number exits loop
            break
    print('You entered', x)
```

Run

```
Please enter a nonnegative integer:3
You entered 3
Please enter a nonnegative integer:-3
Traceback (most recent call last):
   File "nonnegconvert.py", line 8, in <module>
     x = nonnegint(input('Please enter a nonnegative integer:'))
   File "nonnegconvert.py", line 4, in nonnegint
     raise ValueError(result)
ValueError: -3
```

- The ***non_neg_int*** function does not accept any negative integer; hence, it raises a ***ValueError*** exception.

Raising Exceptions

- The following program catches the exception so the program does not crash.

```
def non_neg_int(n):
    """ Converts argument n into a nonnegative integer, if possible.
    Raises a ValueError if the argument is not convertible
    to a nonnegative integer. """
    result = int(n)
    if result < 0:
        raise ValueError(result)
    return result
while True:
    try:
        x = non_neg_int(input('Please enter a nonnegative integer:'))
        if x == 999: # Secret number exits loop
            break
        print('You entered', x)
    except ValueError:
            print('The value you entered is not acceptable')
```

Run

```
Please enter a nonnegative integer:3
You entered 3
Please enter a nonnegative integer:-5
The value you entered is not acceptable
Please enter a nonnegative integer:5
You entered 5
Please enter a nonnegative integer:999
```


Raising Exceptions

- If none of Python's built-in exception types is appropriate to describe the exception you need to raise, you can use the generic ***Exception class*** and provide a descriptive message to its constructor, as in
  raise Exception('Cannot add non-integer to restricted list')
- If raised and uncaught, the interpreter will print the following line at the end of the stack trace:
  Exception: Cannot add non-integer to restricted list
- Later we will see a better way to customize exceptions by designing our own custom exception classes that integrate seamlessly with Python's exception handling infrastructure.

Re-Raising Exceptions

- Sometimes it is appropriate for a function (or method) to catch an exception, take some action appropriate to its local context, and then ***re-raise*** the same exception so that the function's caller can take further action if necessary.
- **Example:** In the following program, the ***count_elements*** function accepts a list, ***lst***, presumed to contain only integers, and a ***Boolean function predicate***.
- The ***predicate function*** parameter accepts a single argument and returns true or false based on whether or not its argument parameter has a certain property.
- The program defines two such ***predicate functions***:
  - ***is_prime function***, which determines if its integer argument is prime
  - ***non_neg function***, which determines if its argument is a nonnegative integer.

Re-Raising Exceptions

- Both ***is_prime*** and ***non_neg*** can raise a ***TypeError exception*** if the caller passes a non-integer argument.
- Neither of these two functions attempts to handle the ***TypeError exception*** itself.

```
def is_prime(n):
    """ Returns True if nonnegative integer n is prime;
    otherwise, returns false.
    Raises a TypeError exception if n is not an integer. """
    from math import sqrt
    if n == 2:     # 2 is the only even prime number
        return True
    if n < 2 or n % 2 == 0: # Handle simple cases immediately
        return False       # Raises a TypeError if n is not an integer
    trial_factor = 3
    root = sqrt(n) + 1
    while trial_factor <= root:
        if n % trial_factor == 0:     # Is trial factor a factor?
            return False     # Yes, return right away
        trial_factor += 2   # Next potential factor, skip evens
    return True            # Tried them all, must be prime
```

Re-Raising Exceptions

```
def non_neg(n):
    """ Determines if n is nonnegative.
    Raises a TypeError if n is not an integer. """
    return n > 0
def count_elements(lst, predicate):
    """ Counts the number of integers in list lst that are
    acceptable to a given predicate (Boolean function).
    Prints an error message and raises a type error if
    the list contains an element incompatible with
    the predicate. """
    count = 0
    for x in lst:
        try:
            if predicate(x):
                count += 1
        except TypeError:
            print(x, 'is a not an acceptable element')
            raise    # Re-raise the caught exception
    return count
```

The *count_elements* function uses a *try* statement to defend against the possible exceptions raised by *is_prime* and *non_neg*.

If it catches a *TypeError* exception, it prints a diagnostic message alerting the user that it found an element in the list that is not compatible with the predicate’s expected parameter type.

Re-Raising Exceptions

```
def main():
    print(count_elements([3, -71, 22, -19, 2, 9], non_neg))
    print(count_elements([2, 3, 4, 5, 6, 8, 9], is_prime))
    print(count_elements([2, 4, '6', 8, 'x', 7], is_prime))
if __name__ == '__main__':
    main()
```

Run

```
4
3
6 is a not an acceptable element
Traceback (most recent call last):
  File "C:/My Python Programs/reraise.py", line 42, in <module>
    main()
  File "C:My Python Programs/reraise.py", line 40, in <module>
    print(count_elements([2, 4, '6', 8, 'x', 7], is_prime))
  File "C:/My Python Programs/reraise.py", line 31, in <module>
    if predicate(x):
  File "C:My Python Programs/reraise.py", line 9, in <module>
    if n < 2 or n % 2 == 0: # Handle simple cases immediately
builtins.TypeError: '<' not supported between instances of 'str' and 'int'
```

Re-Raising Exceptions

- As we can see, the program prints the error message from the ***except block*** in ***count_elements***, and then terminates due to handler's re-raising of the exception it caught.
- After catching the TypeError exception and reporting the problem, the ***count_elements*** function re-raises the same exception for the benefit of its caller via the statement

```
raise       # Re-raises the same exception it caught
```

- The ***raise*** keyword appearing by itself within an except block will re-raise the same exception object that the block caught.
- If we wrap the calling code of the above program with a try, as in:

```
# Wrap the calling code in a try/except statement
def main():
  try:
    print(count_elements([3, -71, 22, -19, 2, 9], non_neg))
    print(count_elements([2, 3, 4, 5, 6, 8, 9], is_prime))
    print(count_elements([2, 4, '6', 8, 'x', 7], is_prime))
  except TypeError:
    print('Error in count_elements')
```

11

Re-Raising Exceptions

Run

```
4
3
6 is a not an acceptable element
Error in count_elements
```

- What is the reason for re-raising an exception?
- After all, the count_elements function could just print the message and continue.
- If it does so, however, the count that it eventually returns would be meaningless, and its caller would not know that count_elements had a problem.
- Re-raising the exception enables count_elements's caller to be informed of the failure so the caller can react to the exception in its own way.

Re-Raising Exceptions

- In general, suppose some function A calls function B that calls function C. The call chain thus looks like

  $$A \rightarrow B \rightarrow C$$

- If C raises an exception, functions A and B both may need to know about it to take appropriate action.
- Function B is closer to C in the call chain. B can catch the exception raised by C, remedy the situation as best it can, and then ensure that its caller (A) receives the same exception. A then can take action appropriate to its own context.
- The idea is that B is the caller closest to the exception origin (C), and B has information unique to its context that its caller (A) would not have. B should handle any exceptions it expects.

Re-Raising Exceptions

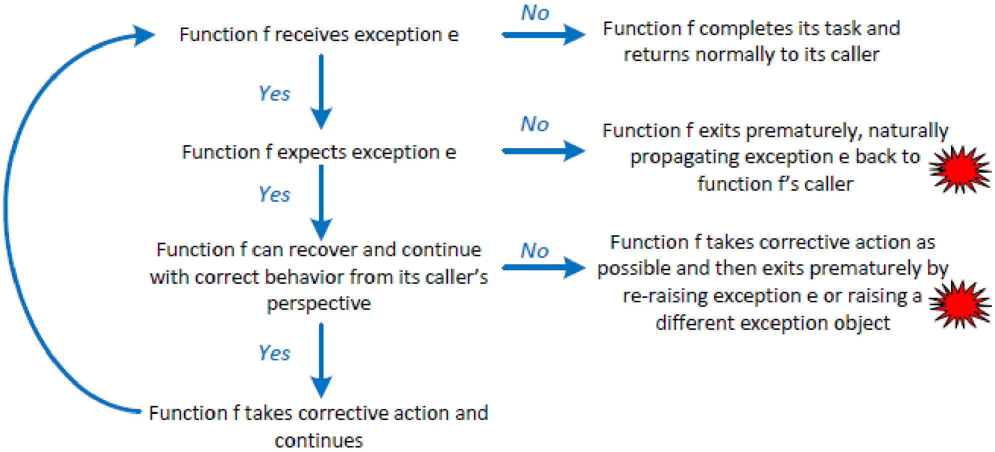
- If the exception is such that B can repair the situation, continue its execution, and in the end correctly fulfill A’s expectations, then there is no reason for B to re-raise the exceptions it caught from C.
- On the other hand, if C’s exception makes B unable to meet A’s expectations, B can do local damage control, but it must also raise an exception that A can process.
- Often this means re-raising the same exception, or raising a different exception that is more B-specific.
- What if C raises an exception type that B does not expect? This means B has no except block to handle that type of exception.
- In this case, the exception propagates naturally back up to A, and it then becomes A responsibility to deal with it.

Re-Raising Exceptions

- This technique represents a good rule of thumb for managing exceptions.
- A function (or method) should catch any exceptions it expects, ignore exceptions it does not expect (or cannot handle in some way), and avoid a catch-all exception handler.
- The following figure illustrates this basic exception handling strategy.
- It shows a flowchart of the execution of a function named f as it encounters exceptions.

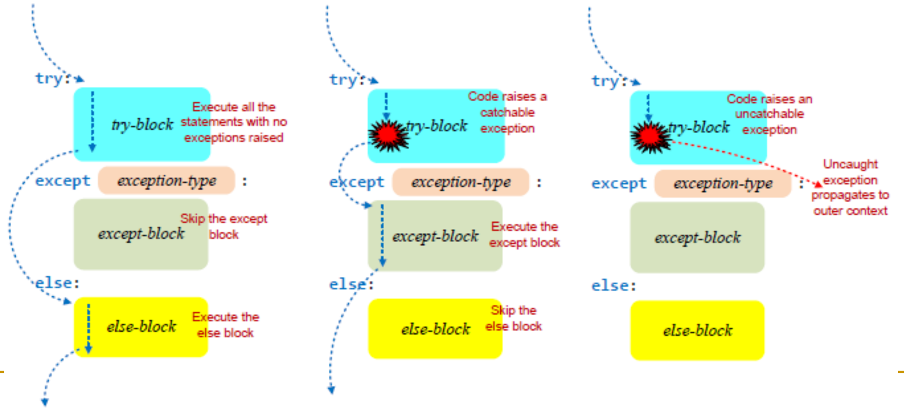
Re-Raising Exceptions

- Function f handles exceptions it expects, potentially repairing what it can and sending an exception to its caller if it cannot.
- Function f ignores exceptions it does not expect or cannot handle.



The try Statement's Optional else Block

- The Python try statement supports an optional ***else*** block.
- If the code within the ***try*** block does not produce an exception, no ***except*** blocks trigger, and the program's execution continues with code in the ***else*** block.
- The figure shows the possible program execution flows within a ***try/else*** statement.



17

The try Statement's Optional else Block

- The ***else*** block, if present, must appear after all of the except blocks.
- Since the code in the else block executes only if the code in the try block does not raise an exception, why not just append the code in the else block to the end of the code within the *try* block and eliminate the *else* block altogether?
- The code restructured in this way may not behave identically to the original code.
- The following program demonstrates the different behavior.

```
def fun1():
    try:
        print('try code')
    except:
        print('exception handling code')
    else:
        print('no exception raised code')
        x = int('a') # Raises an exception
```

Science Minia University

18

The try Statement's Optional else Block

```python
def fun2():
    try:
        print('try code')
        print('no exception raised code')
        x = int('a') # Raises an exception
    except:
        print('exception handling code')
print('Calling fun2')
fun2()
print('-------------')
print('Calling fun1')
fun1()
```

- The ***fun1*** function uses an ***else*** block, and ***fun2*** moves the ***else*** code up into the ***try*** block.
- If the code in the original ***else*** block can raise an exception, moving it up into the ***try*** block means that one of the ***try*** statement's ***except*** blocks could catch that exception.

The try Statement's Optional else Block

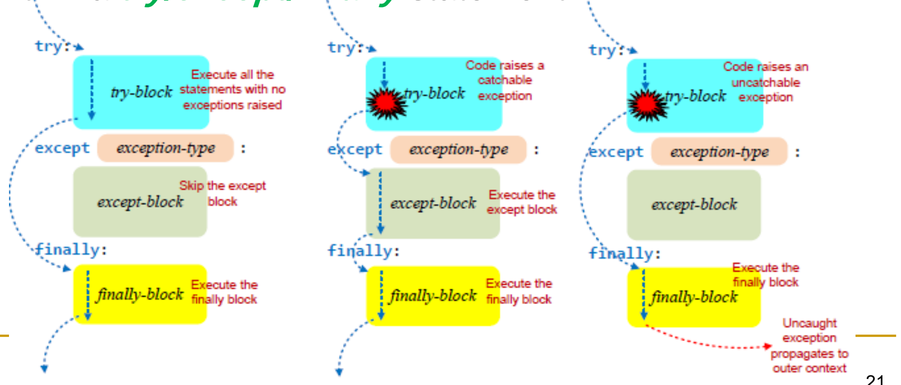
- Leaving the code in the ***else*** block means that any exception it might raise cannot be caught by one of that ***try*** statement's except blocks.

Run

```
Calling fun2
try code
no exception raised code
exception handling code
-------------
Calling fun1
try code
no exception raised code
Traceback (most recent call last):
  File "C:/My Python Programs/tryElse.py", line 20, in <module>
    fun1()
  File "C:/My Python Programs/tryElse.py", line 8, in <module>
    x = int('a') # Raises an exception
builtins.ValueError: invalid literal for int() with base 10: 'a'
```

finally block

- A *try* statement may include an optional *finally* block.
- Code within a *finally* block always executes whether the *try* block raises an exception or not.
- A *finally* block usually contains "clean-up code" that must execute due to activity initiated in the *try* block.
- The figure shows the possible program execution flows within a *try/except/finally* statement.



No exceptions Caught exception Uncaught exception

21

finally block

- Consider the following program that opens a file and reads its contents.

```
# Sum the values in a text file containing integer values
try:
   f = open('mydata.dat')
except OSError:
   print('Could not open file')
else: # File opened properly
   sum = 0
   try:
      for line in f:
         sum += int(line)
      f.close() # Close the file if no exception
   except Exception as er:
      print(er) # Show the problem
   f.close() # Close the file if exception
print('sum =', sum)
```

mydata.dat

```
5
2
3
```

Output

```
sum = 10
```

mydata.dat

```
5
2
four
3
```

Output

```
nvalid literal for int() with base 10: 'four\n'
sum = 7
```

finally block

- The above program uses two ***try*** statements.
- The first try statement defends against an ***OSError*** exception.
- The operating system will prompt the open function to raise such an exception if it cannot satisfy the request; for example, the file may not exist in the current directory or the user may not have sufficient permissions to access the file.
- The program does not proceed if it cannot open the file for reading.
- If no file named ***mydata.dat*** exists in the current directory, the program will print: **Could not open file**
- The code in the outer ***try*** statement's else block contains the interesting part.

finally block

- Once the file is open, more exceptions are possible. In particular, the text file may contain a string that is does not evaluate to a number.
- The inner try statement includes a catch-all except block to handle any exception that may arise.
- Observe that the above program contains two identical statements to close the file:
  - If the execution makes it all the way to the end of the inner try block, it needs to close the file.
  - If an exception arises in the inner try block, the exception handler must close the file.
- This code duplication is undesirable, so we can use a finally block to consolidate the code, as shown in the following program:

finally block

```
# Sum the values in a text file containing integers
try:
    f = open('mydata.dat')
except OSError:
    print('Could not open file')
else:
    sum = 0
    try:
        for line in f:
            sum += int(line)
    except Exception as er:
        print(er) # Show the problem
    finally:
        f.close() # Close the file
    print('sum =', sum)
```

- As we know, the ***with/as*** statement takes care of the details of properly closing a file should an exception arise.
- But, the ***with/as*** statement will not automatically handle any exceptions.

finally block

- We can remedy this with another pair of nested ***try*** statements, as the following program shows:

```
# Sum the values in a text file containing integers
try:
    with open('mydata.dat') as f:
        sum = 0
        try:
            for line in f:
                sum += int(line)
        except Exception as er:
            print(er) # Show the problem
        print('sum =', sum)
except OSError:
    print('Could not open file')
```

- Since this program uses the ***with/as*** statement, the program does not explicitly call the file object's ***close*** method.
- The omission of the ***f.close()*** statement eliminates the need of the ***finally*** block.

Custom Exceptions

- Now we are familiar with inheritance and Python's exception handling, so we can create our own custom exception types.
- Consider the following interactive session:

```
>>> v = ValueError()
>>> v
ValueError()
>>> isinstance(v, ValueError)
True
>>> isinstance(v, Exception)
True
```

- We can see that ***ValueError*** is a subclass of ***Exception***.
- Almost every standard exception class is a direct or indirect subclass of ***Exception***.
- Amongst the standard exception classes, only *BaseException*, *KeyboardInterrupt*, *SystemExit*, and *GeneratorExit* are not related via inheritance to the ***Exception*** class.

Custom Exceptions

- Due to the ***is a*** relationship imposed by inheritance, any exception object of a class other than the four excluded types mentioned above is compatible with the ***Exception*** type.
- This explains why we use ***Exception*** as the ***"catch-all" exception*** type in an exception handler.
- Even though Python provides 68 standard exception classes, we may not find one that exactly meets our needs.
- Inheritance makes it easy to define our own custom exception types.
- **Example:** Consider the program that defines a simple stopwatch timer class.
- Suppose we wish to consider an attempt to stop a non-running stopwatch an error worthy of an exception.

Custom Exceptions

- What we need is a ***StopwatchException*** class.
- Making such a class turns out to be remarkably easy; the following class meets the requirement:

```
class StopwatchException(Exception):
    pass
```

- The ***StopwatchException*** class inherits from ***Exception***, but its empty class body means that ***StopwatchException*** adds nothing to what the ***Exception*** class already offers.
- The value that ***StopwatchException*** adds is this:
  - it defines a new exception type that can participate in Python's exception handling infrastructure.
- Because we derived ***StopwatchException*** from ***Exception***, a ***"catch-all" except block*** will catch a ***StopwatchException*** object even if a ***try statement*** lacks an explicit StopwatchException-specific except block.

Custom Exceptions

- We can rewrite the ***Stopwatch.stop*** method in class Stopwatch:

```
# Other details about the class omitted ...
def stop(self):
    """ Stops the stopwatch, unless it is not running.
    Updates the accumulated elapsed time. """
    if self._running:
            self._elapsed = perf_counter() - self._start_time
            self._running = False # Clock stopped
    else:
            raise StopwatchException()
```

- Exception handling code now can distinguish between a *stopwatch error* and a *ValueError*.

```
try:
    # Some code that may raise a StopwatchException or
    # ValueError exception
except ValueError:
    pass # Add code to process ValueError
except StopwatchException:
    pass # Add code to process StopwatchException
except Exception:
    pass # Add code to process all other normal exceptions
```