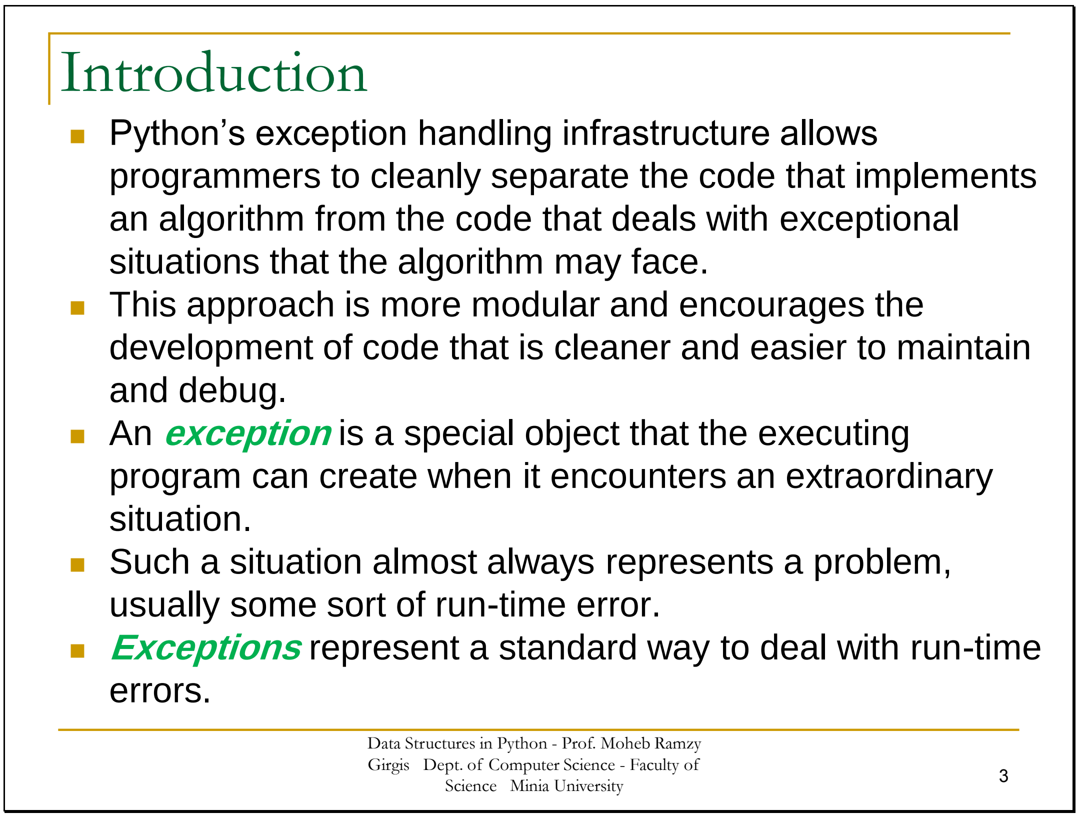
Data Structures in Python

4. Exception Handling (I)

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

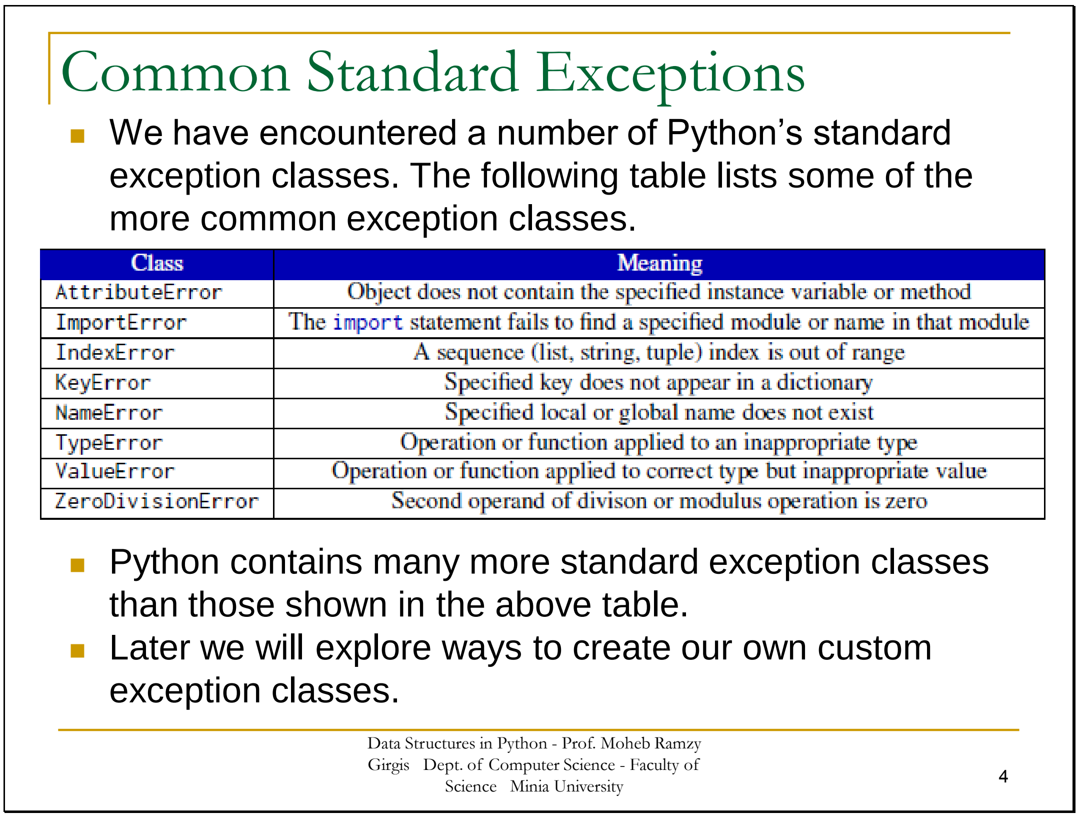
Introduction

- In our programming experience so far we have encountered several kinds of run-time exceptions, such as division by zero, accessing a list with an out-of-range index, and attempting to convert a non-number to an integer.
- We have seen these and other run-time exceptions immediately terminate a running program.
- Python provides a standard mechanism called exception handling that allows programmers to deal with these kinds of run-time exceptions and many more.
- Rather than always terminating the program's execution, an executing program can detect the problem when it arises and possibly execute code to correct the issue or mitigate it in some way.
- This lecture explores handling exceptions in Python.



Introduction

- Python's exception handling infrastructure allows programmers to cleanly separate the code that implements an algorithm from the code that deals with exceptional situations that the algorithm may face.
- This approach is more modular and encourages the development of code that is cleaner and easier to maintain and debug.
- An ***exception*** is a special object that the executing program can create when it encounters an extraordinary situation.
- Such a situation almost always represents a problem, usually some sort of run-time error.
- ***Exceptions*** represent a standard way to deal with run-time errors.



Common Standard Exceptions

- We have encountered a number of Python's standard exception classes. The following table lists some of the more common exception classes.

Class	Meaning
`AttributeError`	Object does not contain the specified instance variable or method
`ImportError`	The `import` statement fails to find a specified module or name in that module
`IndexError`	A sequence (list, string, tuple) index is out of range
`KeyError`	Specified key does not appear in a dictionary
`NameError`	Specified local or global name does not exist
`TypeError`	Operation or function applied to an inappropriate type
`ValueError`	Operation or function applied to correct type but inappropriate value
`ZeroDivisionError`	Second operand of divison or modulus operation is zero

- Python contains many more standard exception classes than those shown in the above table.
- Later we will explore ways to create our own custom exception classes.

Handling Exceptions

- The following program computes the quotient of two integer values supplied by the user.

```
# Get two integers from the user
print('Please enter two numbers to divide.')
num1 = int(input('Please enter the dividend: '))
num2 = int(input('Please enter the divisor: '))
print('{0} divided by {1} = {2}'.format(num1, num2, num1/num2))
```

- This program works fine until the user attempts to divide by zero. In this case the program execution produces a ***ZeroDivisionError*** exception.

Run

```
Please enter the dividend: 4
Please enter the divisor: 0
Traceback (most recent call last):
  File "C:/Python/My Python Programs/divideNumbers.py",
    line 5, in <module>
    print('{0} divided by {1} = {2}'.format(num1, num2, num1/num2))
builtins.ZeroDivisionError: division by zero
```

Handling Exceptions

- We can defend against the ***ZeroDivisionError*** exception with a conditional statement, as follows:

```
# Get two integers from the user
print('Please enter two numbers to divide.')
num1 = int(input('Please enter the dividend: '))
num2 = int(input('Please enter the divisor: '))
if num2 != 0:
   print('{0} divided by {1} = {2}'.format(num1, num2, num1/num2))
else:
   print('Cannot divide by zero')
```

- This solution works well for code like that shown above.
- Consider the following program that asks the user for a small integer value.

```
val = int(input("Please enter a small positive integer: "))
print('You entered', val)
```

Run

```
Please enter a small positive integer: 5
You entered 5
```

Handling Exceptions

- A user easily and innocently can ruin the programmer's original intentions, as the following sample run illustrates:

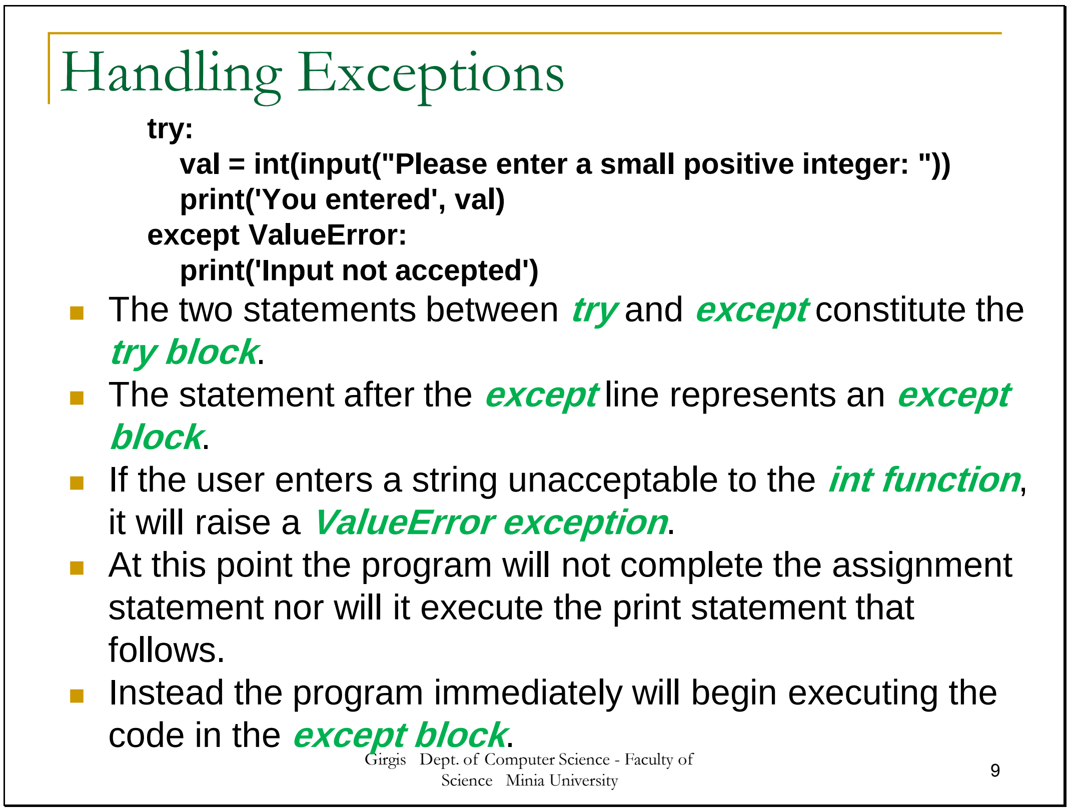
Run

```
Please enter a small positive integer: five
Traceback (most recent call last):
  File "C:/Python/My Python Programs/divideNumbers.py",
    line 7, in <module>
    val = int(input("Please enter a small positive integer: "))
builtins.ValueError: invalid literal for int() with base 10: 'five'
```

- For human, the response five should be just as acceptable as 5. But, the strings acceptable to the Python ***int*** function, can contain only numeric characters and an optional leading sign character (+ or -).
- The user's input causes the program to produce a run-time exception (***ValueError exception***)
- The program reacts to the exception by printing a message and terminating itself.

Handling Exceptions

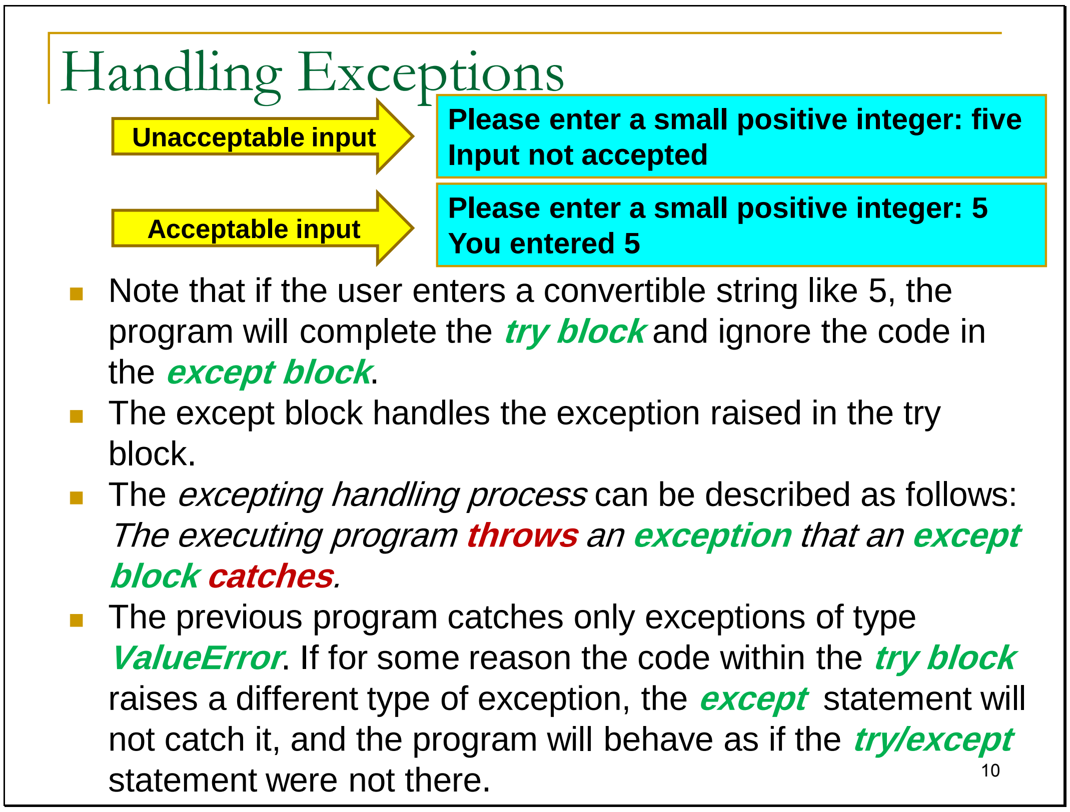
- Unfortunately, attempting to add a check for the user's input as we did with the division program is not easy.
- We basically need to determine if the arbitrary string the user enters is acceptable to the ***int conversion function***.
- The string must contain only the digit characters '0', '1', '2', '3', '4', '5', '6', '7', '8', or '9', and it may contain a leading '-' or '+' character indicating the number's sign.
- The best approach is to execute the potentially problematic code within a ***try statement***.
- If the code raises an exception, the program's execution does not necessarily terminate; instead, the program's execution jumps immediately to a different block within the ***try statement***.
- The following program wraps the code from the previous program within a try statement to successfully defend again bad user input.



Handling Exceptions

```
try:
   val = int(input("Please enter a small positive integer: "))
   print('You entered', val)
except ValueError:
   print('Input not accepted')
```

- The two statements between ***try*** and ***except*** constitute the ***try block***.
- The statement after the ***except*** line represents an ***except block***.
- If the user enters a string unacceptable to the ***int function***, it will raise a ***ValueError exception***.
- At this point the program will not complete the assignment statement nor will it execute the print statement that follows.
- Instead the program immediately will begin executing the code in the ***except block***.



Handling Exceptions

Unacceptable input

```
Please enter a small positive integer: five
Input not accepted
```

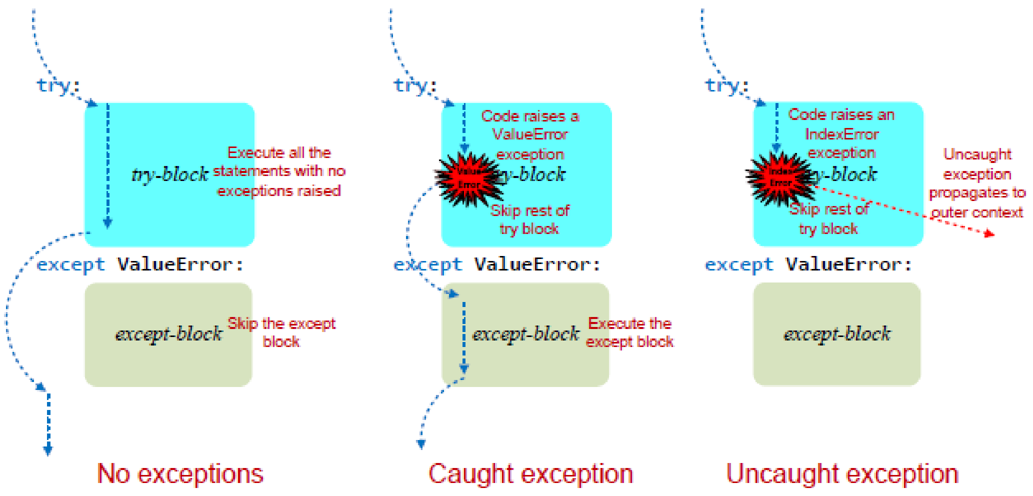
Acceptable input

```
Please enter a small positive integer: 5
You entered 5
```

- Note that if the user enters a convertible string like 5, the program will complete the ***try block*** and ignore the code in the ***except block***.
- The except block handles the exception raised in the try block.
- The *excepting handling process* can be described as follows: *The executing program* ***throws*** *an* ***exception*** *that an* ***except block catches***.
- The previous program catches only exceptions of type ***ValueError***. If for some reason the code within the ***try block*** raises a different type of exception, the ***except*** statement will not catch it, and the program will behave as if the ***try/except*** statement were not there.

Handling Exceptions

- The following figure contrasts the possible program execution flows within a ***try/except statement***.



Handling Multiple Exceptions

- A try statement can have multiple except blocks. Each except block must catch a different type of exception object.
- The following program offers three except blocks. Its try statement specifically can catch ValueError, IndexError, and ZeroDivisionError exceptions.

```
import random
for i in range(10): # Loop 10 times
   print('Beginning of loop iteration', i)
   try:
      r = random.randint(1, 3) # r is pseudorandomly 1, 2, or 3
      if r == 1:
         print(int('Fred')) # Try to convert a non-integer
      elif r == 2:
         [][2] = 5 # Try to assign to a nonexistent index of the empty list
      else:
         print(3/0) # Try to divide by zero
```

Handling Multiple Exceptions

```
except ValueError:
    print('Cannot convert integer')
except IndexError:
    print('List index is out of range')
except ZeroDivisionError:
    print('Division by zero not allowed')
print('End of loop iteration', i)
```

The expression [] represents the empty list. The expression [][2] represents the element at index 2 within the empty list (there is no such element).

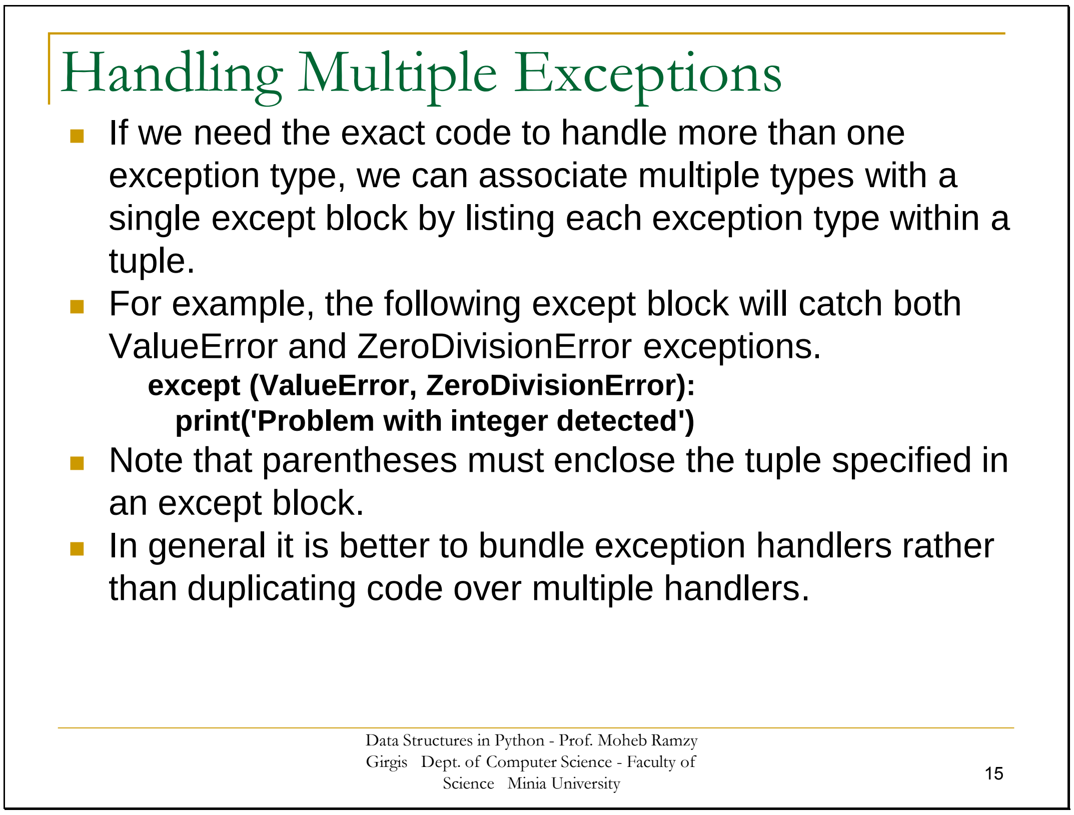
- Each time through the loop the code within the try block of the program will raise one of three different exceptions based on the generated pseudorandom number.
- The program offers three except blocks.
- If the code in the try block raises one of the three types of exceptions, the program will execute the code in the matching except block.
- Only code in one of the three except blocks will execute as a result of the exception.

Handling Multiple Exceptions

- A sample run is shown below. Observe that only one except block executes each time through the loop.

```
Beginning of loop iteration 0
Division by zero not allowed
End of loop iteration 0
Beginning of loop iteration 1
List index is out of range
End of loop iteration 1
Beginning of loop iteration 2
Cannot convert integer
End of loop iteration 2
Beginning of loop iteration 3
Division by zero not allowed
End of loop iteration 3
Beginning of loop iteration 4
Division by zero not allowed
End of loop iteration 4
```

```
Beginning of loop iteration 5
Cannot convert integer
End of loop iteration 5
Beginning of loop iteration 6
List index is out of range
End of loop iteration 6
Beginning of loop iteration 7
List index is out of range
End of loop iteration 7
Beginning of loop iteration 8
Cannot convert integer
End of loop iteration 8
Beginning of loop iteration 9
List index is out of range
End of loop iteration 9
```

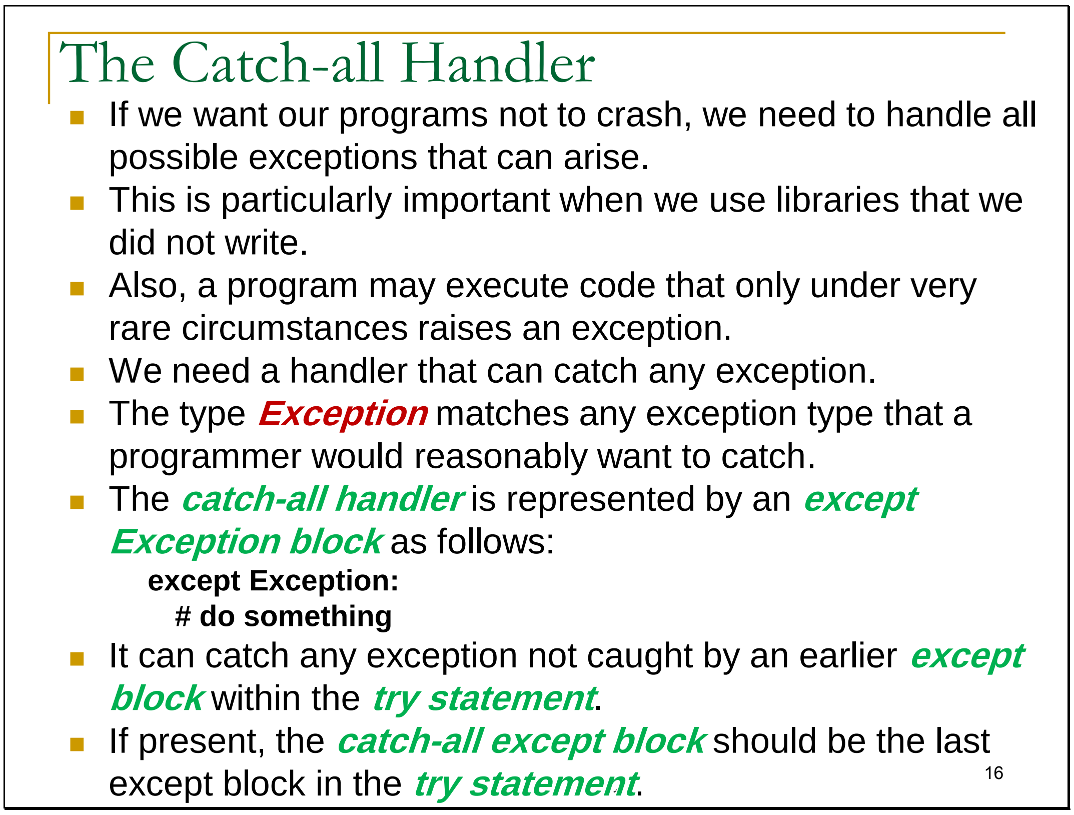


Handling Multiple Exceptions

- If we need the exact code to handle more than one exception type, we can associate multiple types with a single except block by listing each exception type within a tuple.
- For example, the following except block will catch both ValueError and ZeroDivisionError exceptions.

```
except (ValueError, ZeroDivisionError):
    print('Problem with integer detected')
```

- Note that parentheses must enclose the tuple specified in an except block.
- In general it is better to bundle exception handlers rather than duplicating code over multiple handlers.

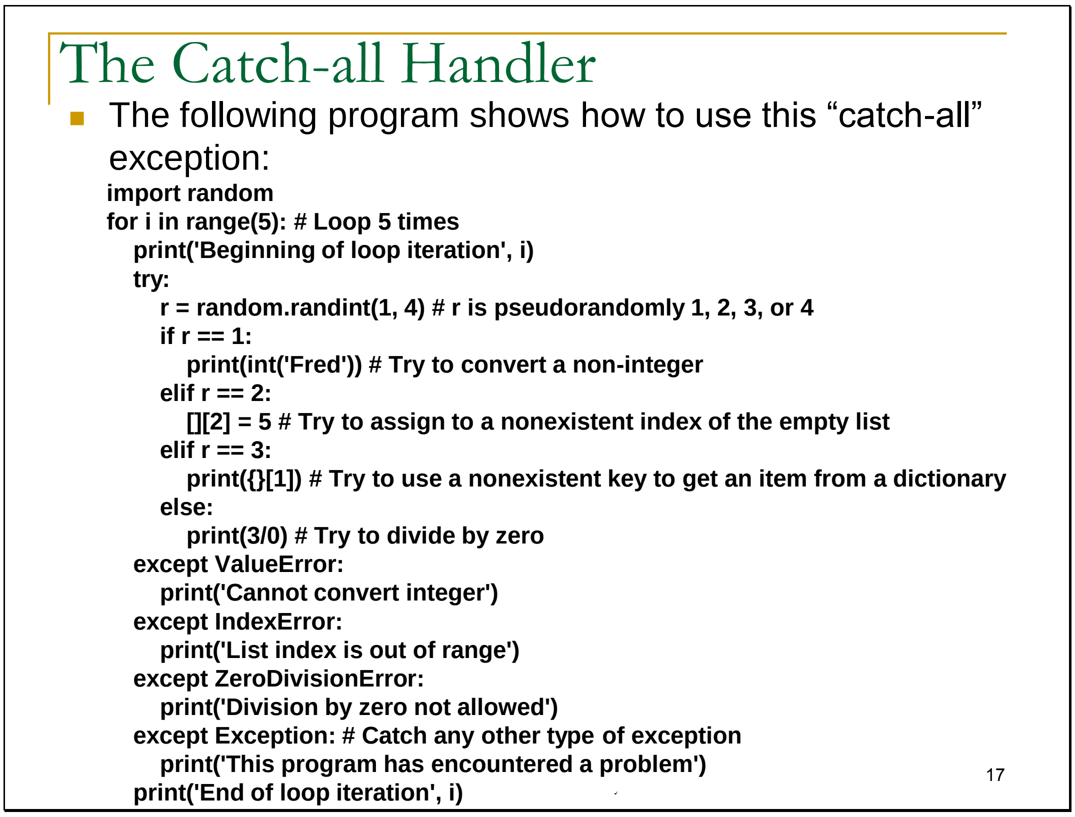


The Catch-all Handler

- If we want our programs not to crash, we need to handle all possible exceptions that can arise.
- This is particularly important when we use libraries that we did not write.
- Also, a program may execute code that only under very rare circumstances raises an exception.
- We need a handler that can catch any exception.
- The type ***Exception*** matches any exception type that a programmer would reasonably want to catch.
- The ***catch-all handler*** is represented by an ***except Exception block*** as follows:

```
except Exception:
    # do something
```

- It can catch any exception not caught by an earlier ***except block*** within the ***try statement***.
- If present, the ***catch-all except block*** should be the last except block in the ***try statement***.

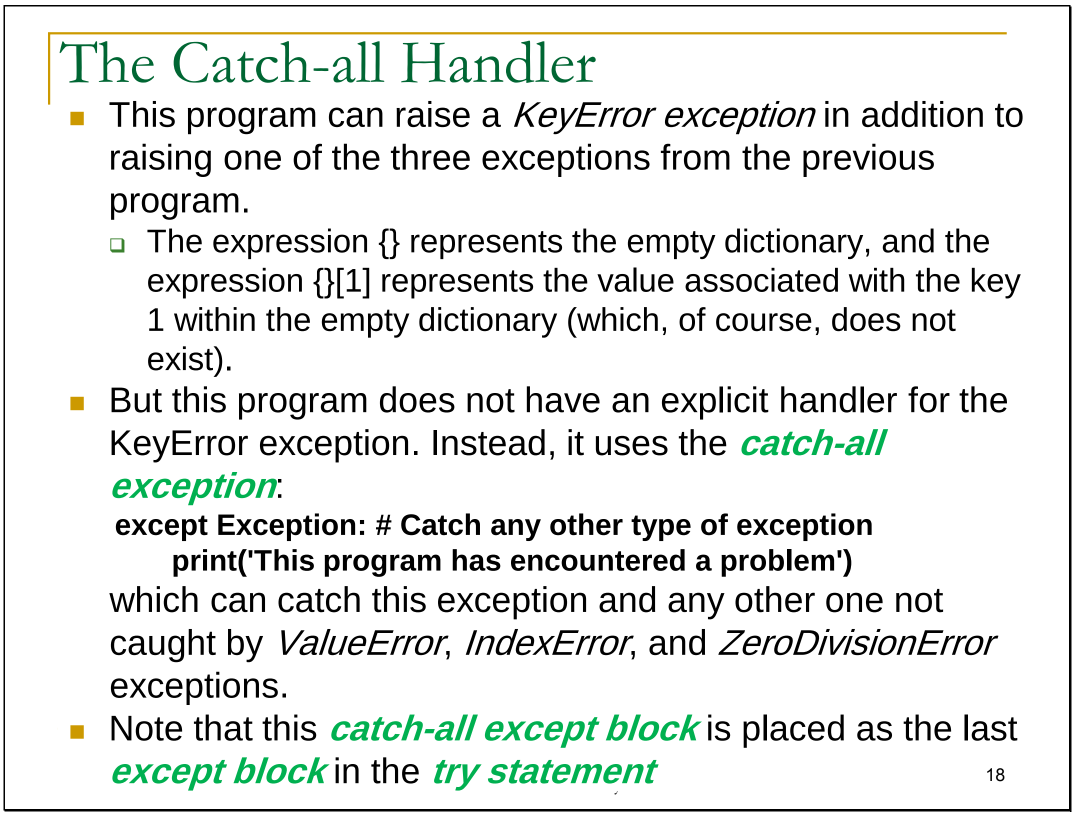


The Catch-all Handler

- The following program shows how to use this “catch-all” exception:

```
import random
for i in range(5): # Loop 5 times
   print('Beginning of loop iteration', i)
   try:
      r = random.randint(1, 4) # r is pseudorandomly 1, 2, 3, or 4
      if r == 1:
         print(int('Fred')) # Try to convert a non-integer
      elif r == 2:
         [][2] = 5 # Try to assign to a nonexistent index of the empty list
      elif r == 3:
         print({}[1]) # Try to use a nonexistent key to get an item from a dictionary
      else:
         print(3/0) # Try to divide by zero
   except ValueError:
      print('Cannot convert integer')
   except IndexError:
      print('List index is out of range')
   except ZeroDivisionError:
      print('Division by zero not allowed')
   except Exception: # Catch any other type of exception
      print('This program has encountered a problem')
   print('End of loop iteration', i)
```

17



The Catch-all Handler

- This program can raise a *KeyError exception* in addition to raising one of the three exceptions from the previous program.
  - The expression {} represents the empty dictionary, and the expression {}[1] represents the value associated with the key 1 within the empty dictionary (which, of course, does not exist).
- But this program does not have an explicit handler for the KeyError exception. Instead, it uses the ***catch-all exception***:

  ```
  except Exception: # Catch any other type of exception
        print('This program has encountered a problem')
  ```

  which can catch this exception and any other one not caught by *ValueError*, *IndexError*, and *ZeroDivisionError* exceptions.
- Note that this ***catch-all except block*** is placed as the last ***except block*** in the ***try statement***

18

The Catch-all Handler

- There are a few exceptions that the ***Exception*** type does not match, such as:
  - The call sys.exit(0), which raises an exception and terminates the executing program
  - the keyboard interrupt (Ctrl C) entered by the user.

Sample Run

```
Beginning of loop iteration 0
This program has encountered a problem
End of loop iteration 0
Beginning of loop iteration 1
Cannot convert integer
End of loop iteration 1
Beginning of loop iteration 2
This program has encountered a problem
End of loop iteration 2
Beginning of loop iteration 3
Division by zero not allowed
End of loop iteration 3
Beginning of loop iteration 4
List index is out of range
End of loop iteration 4
```

- The ***try statement*** allows us to catch such exceptions and any previously uncaught exception with an ***untyped except block***:

```
except:
   # do something
```

Catching Exception Objects

- The ***except*** blocks "catch" ***exception objects***.
- We can inspect the object that an ***except*** block catches if we specify the object's name with the ***as*** keyword.
- The following program shows how to use the ***as*** keyword to get access to the exception object raised by code in the try block.

```
import random
for i in range(5): # Loop 5 times
   print('Beginning of loop iteration', i)
   try:
      r = random.randint(1, 4) # r is pseudorandomly 1, 2, 3, or 4
      if r == 1:
         print(int('Fred')) # Try to convert a non-integer
      elif r == 2:
         [][2] = 5 # Try to assign to a nonexistent index of the empty list
      elif r == 3:
         print({}[1]) # Try to use a nonexistent key to get an item from a dictionary
      else:
         print(3/0) # Try to divide by zero
```

Catching Exception Objects

```
except ValueError as e:
    print('Problem with value ==>', type(e), e)
except IndexError as e:
    print('Problem with list ==>', type(e), e)
except ZeroDivisionError as e:
    print('Problem with division ==>', type(e), e)
except Exception as e:
    print('Problem with something ==>', type(e), e)
print('End of loop iteration', i)
```

The following sample run reveals that each *exception object* prints a message that is meaningful for its particular exception when sent to the print function**.**

```
Beginning of loop iteration 0
Problem with value ==> <class 'ValueError'> invalid literal for int() with base 10: 'Fred'
End of loop iteration 0
Beginning of loop iteration 1
Problem with list ==> <class 'IndexError'> list assignment index out of range
End of loop iteration 1
Beginning of loop iteration 2
Problem with division ==> <class 'ZeroDivisionError'> division by zero
End of loop iteration 2
Beginning of loop iteration 3
Problem with division ==> <class 'ZeroDivisionError'> division by zero
End of loop iteration 3
Beginning of loop iteration 4
Problem with something ==> <class 'KeyError'> 1
End of loop iteration 4
```

21

Exception Handling Scope

- The exception handling examples we have seen so far have been simple programs where all the code is in the main executing module. The origin of the exception is close to the code we can see.
- These examples have not demonstrated the true power of Python’s exceptions.
- To get a better idea of the scope of exceptions, consider the following program:

```
def get_int_in_range(low, high):
    """ Obtains an integer value from the user. Acceptable values
    must fall within the specified range low...high. """
    val = int(input()) # Can raise a ValueError
    while val < low or val > high:
        print('Value out of range, please try again:', end=' ')
        val = int(input()) # Can raise a ValueError
    return val
```

Data Structures in Python - Prof. Moheb Ramzy Girgis Dept. of Computer Science - Faculty of Science Minia University

22

Exception Handling Scope

```
def create_list(n, min, max):
    """ Allows the user to create a list of n elements consisting
    of integers in the range min...max """
    result = []
    while n > 0: # Count down to zero
        print('Enter integer in the range {}...{}:'.format(min, max), end=' ')
        result.append(get_int_in_range(min, max))
        n -= 1
    return result
def main():
    """ Create a list of two elements supplied by the user,
    each element in the range 10...20 """
    lst = create_list(2, 10, 20)
    print(lst)
if __name__ == '__main__':
    main() # Invoke main
```

Sample Normal Run

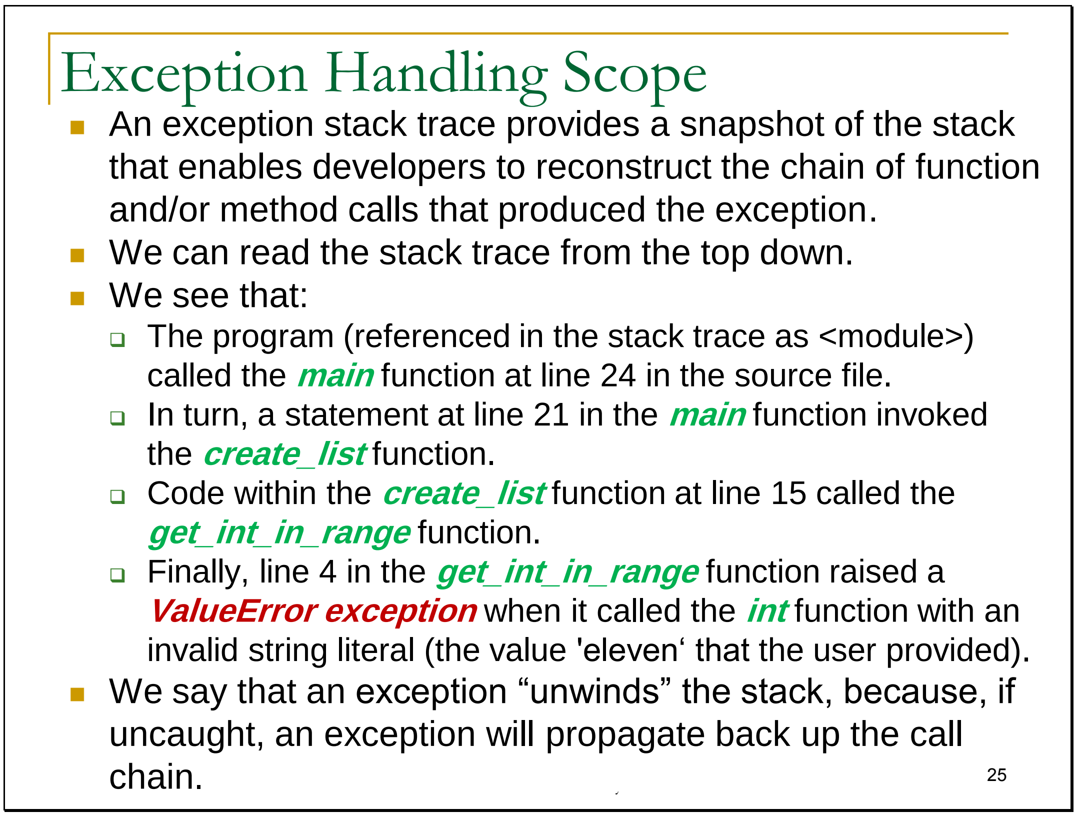
```
Enter integer in the range 10...20: 12
Enter integer in the range 10...20: 15
[12, 15]
```

Exception Handling Scope

- The following shows how the program runs with string input 'eleven' rather than int input:

```
Enter integer in the range 10...20: eleven
Traceback (most recent call last):
  File "C:/My Python Programs/makeIntegerList.py", line 24, in <module>
    main() # Invoke main
  File "C:/My Python Programs/makeIntegerList.py", line 21, in <module>
    lst = create_list(2, 10, 20)
  File "C:/My Python Programs/makeIntegerList.py", line 15, in <module>
    result.append(get_int_in_range(min, max))
  File "C:/My Python Programs/makeIntegerList.py", line 4, in <module>
    #print('Exiting sandbox process')
builtins.ValueError: invalid literal for int() with base 10: 'eleven'
```

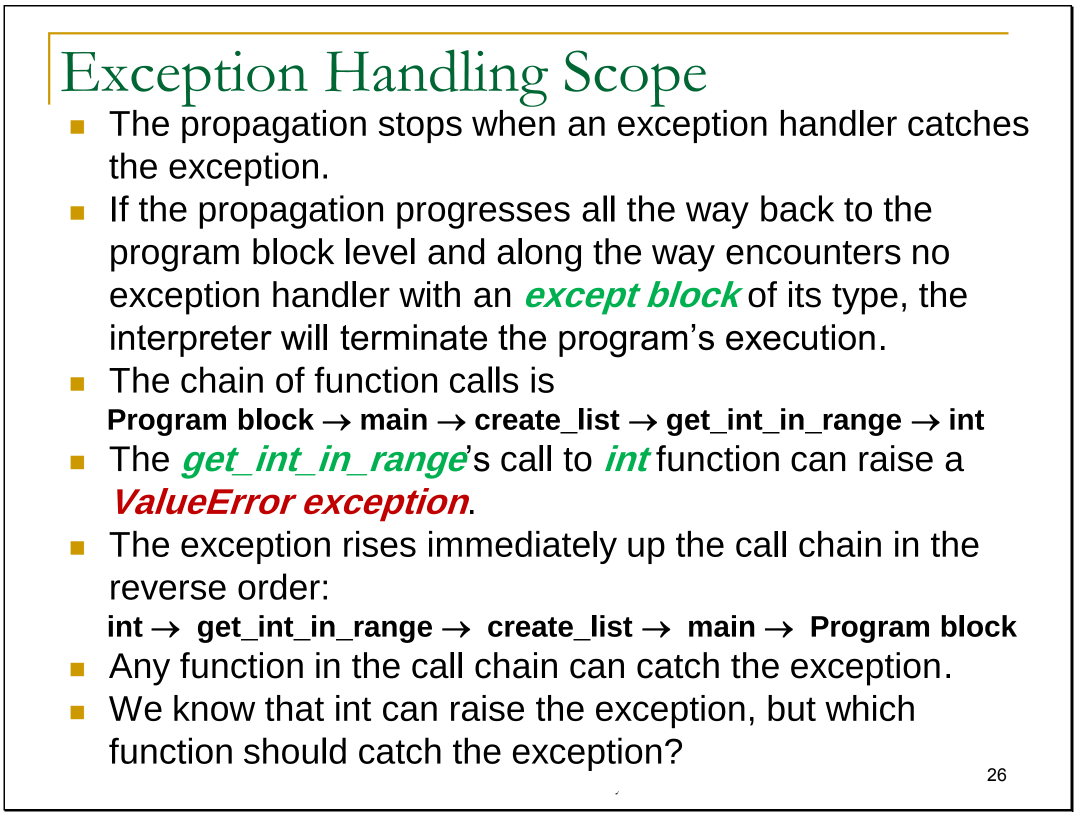
- An uncaught exception produces a ***stack trace***.
- Python uses an area of the computer's memory known as the *stack* to help it control function and method invocations.
- The *stack* stores parameters, return values, and the point in the code where the program's execution should return when a function completes.



Exception Handling Scope

- An exception stack trace provides a snapshot of the stack that enables developers to reconstruct the chain of function and/or method calls that produced the exception.
- We can read the stack trace from the top down.
- We see that:
  - The program (referenced in the stack trace as <module>) called the ***main*** function at line 24 in the source file.
  - In turn, a statement at line 21 in the ***main*** function invoked the ***create_list*** function.
  - Code within the ***create_list*** function at line 15 called the ***get_int_in_range*** function.
  - Finally, line 4 in the ***get_int_in_range*** function raised a ***ValueError exception*** when it called the ***int*** function with an invalid string literal (the value 'eleven‘ that the user provided).
- We say that an exception “unwinds” the stack, because, if uncaught, an exception will propagate back up the call chain.

25



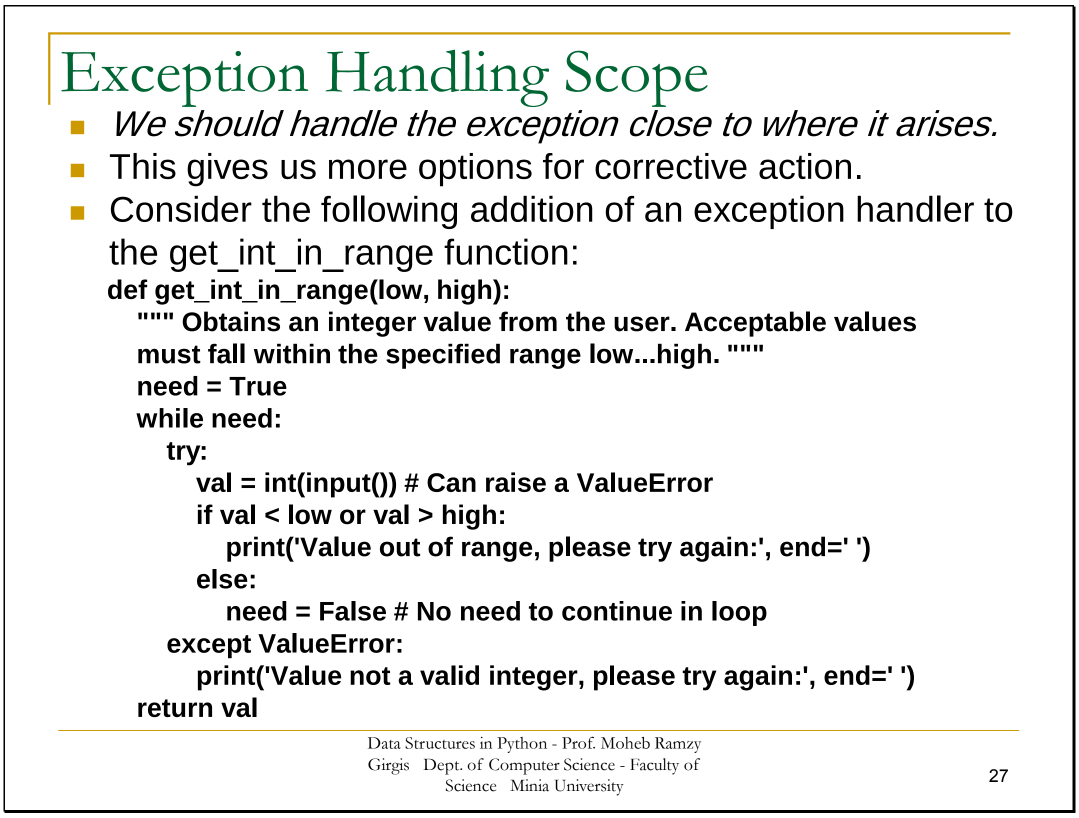
Exception Handling Scope

- The propagation stops when an exception handler catches the exception.
- If the propagation progresses all the way back to the program block level and along the way encounters no exception handler with an ***except block*** of its type, the interpreter will terminate the program’s execution.
- The chain of function calls is

  Program block → main → create_list → get_int_in_range → int
- The ***get_int_in_range***’s call to ***int*** function can raise a ***ValueError exception***.
- The exception rises immediately up the call chain in the reverse order:

  int → get_int_in_range → create_list → main → Program block
- Any function in the call chain can catch the exception.
- We know that int can raise the exception, but which function should catch the exception?

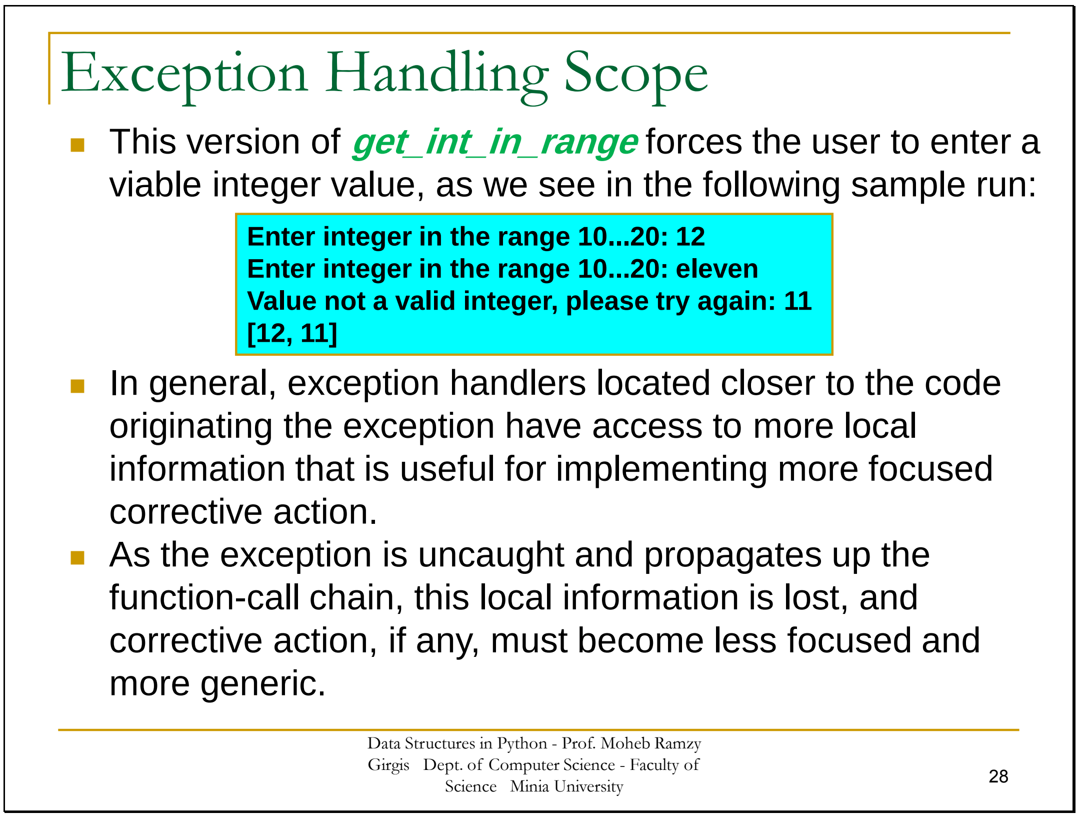
26



Exception Handling Scope

- *We should handle the exception close to where it arises.*
- This gives us more options for corrective action.
- Consider the following addition of an exception handler to the get_int_in_range function:

```
def get_int_in_range(low, high):
    """ Obtains an integer value from the user. Acceptable values
    must fall within the specified range low...high. """
    need = True
    while need:
        try:
            val = int(input()) # Can raise a ValueError
            if val < low or val > high:
                print('Value out of range, please try again:', end=' ')
            else:
                need = False # No need to continue in loop
        except ValueError:
            print('Value not a valid integer, please try again:', end=' ')
    return val
```



Exception Handling Scope

- This version of ***get_int_in_range*** forces the user to enter a viable integer value, as we see in the following sample run:

```
Enter integer in the range 10...20: 12
Enter integer in the range 10...20: eleven
Value not a valid integer, please try again: 11
[12, 11]
```

- In general, exception handlers located closer to the code originating the exception have access to more local information that is useful for implementing more focused corrective action.
- As the exception is uncaught and propagates up the function-call chain, this local information is lost, and corrective action, if any, must become less focused and more generic.