BIOS6642

Exception Handling

Fuyong Xing

Department of Biostatistics and Informatics

Colorado School of Public Health

University of Colorado Anschutz Medical Campus

Outline

- **Handling exceptions**
- Exception handling scope
- Raising exceptions
- The *else* and *finally* blocks

Handling exceptions

- We have seen a few Python's standard exceptions.

```
>>> seq = [2, 5, 11]
>>> print(seq[1])
5
>>> print(seq[3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> print(seq['Fred'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list indices must be integers, not str
```

Handling exceptions

- We have seen a few Python's standard exceptions.

```
>>> d = {}
>>> d['Fred'] = 100
>>> print(d['Fred'])
100
>>> print(d['Freddie'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Freddie'
>>> print(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

Handling exceptions

- We have seen a few Python's standard exceptions.

```
>>> print(int('Fred'))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'Fred'
>>> print(1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Handling exceptions

- We have seen a few Python's standard exceptions.

```
>>> from fractions import Fraction
>>> frac = Fraction(1, 2)
>>> print(frac.numerator)
1
>>> print(frac.numertor)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Fraction' object has no attribute 'numertor'
```

Handling exceptions

- We have seen a few Python's standard exceptions.

```
>>> from fraction import Fraction
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fraction'
>>> from fractions import Fractions
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: cannot import name 'Fractions'
```

Handling exceptions

- Some common standard exceptions

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

Handling exceptions

- When an exception occurs, the program usually terminates (if not properly handled).
- Programs should properly handle exceptions to be robust.

Codes

```
#  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))
```

Output

```
Please enter two numbers to divide.
Please enter the dividend: 4
Please enter the divisor: 0
Traceback (most recent call last):
  File "dividenumbers.py", line 5, in <module>
    print('{0} divided by {1} = {2}'.format(num1, num2, num1/num2))
ZeroDivisionError: division by zero
```

Handling exceptions

- We probably want to use a conditional statement to solve the problem.

Codes

```
#  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')
```

Handling exceptions

- Consider the following program.

Codes

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

Output

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

Handling exceptions

- Consider the following program.

Codes

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

Output

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

How about five?

```
Please enter a small positive integer: five
Traceback (most recent call last):
  File "enterinteger.py", line 1, in <module>
    val = int(input("Please enter a small positive integer: "))
ValueError: invalid literal for int() with base 10: 'five'
```

Handling exceptions

- Can we use the conditional statement to solve the problem?
  - We will need to determine if the arbitrary string the user enters contains only digit characters ('0', '1', …, '9') and potential '-' or '+' characters.
  - It will make the program complicated.

Handling exceptions

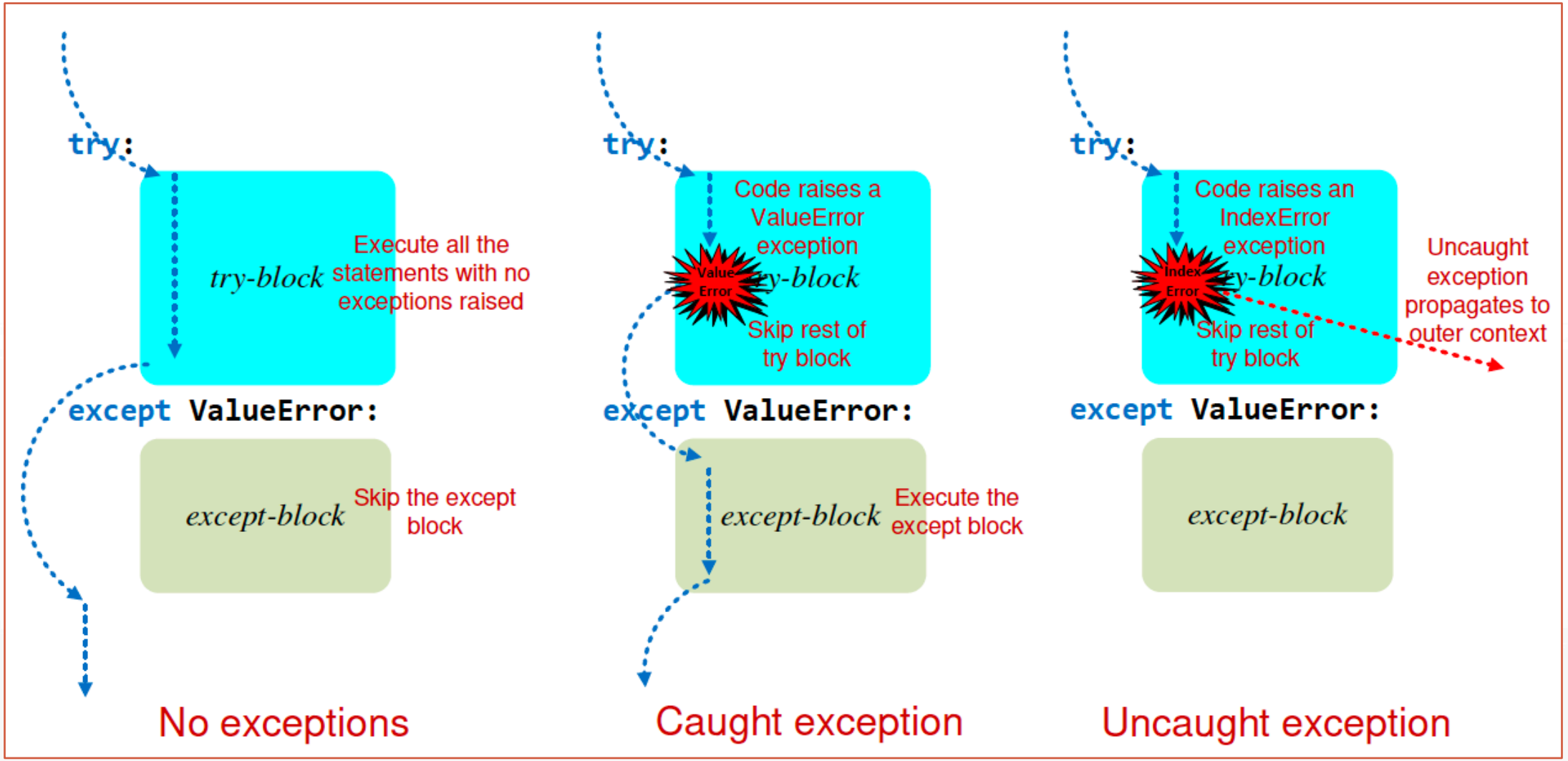
- We can use a *try* statement.

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

- 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.

Handling exceptions

- Execution flow of the *try* statement in the example



Handling exceptions

- The *try* statement works as follows:
  - First, the *try* block (the statement(s) between the *try* and *except* keywords) is executed.
  - If no exception occurs, the *except* block is skipped and execution of the *try* statement is finished.
  - If an exception occurs during execution of the *try* block, the rest of the block is skipped. Then if its type matches the exception named after the *except* keyword, the *except* block is executed, and then execution continues after the *try* statement.
  - If an exception occurs which does not match the exception named in the *except* block, it is passed on to outer *try* statements; if no handler is found, it is an unhandled exception and execution stops.

https://docs.python.org/3/tutorial/errors.html

Handling exceptions

- The previous program only handles one type of exception (i.e., ValueError). It will terminate with an error message if other types of exceptions occur.

Codes

```
try:
    val = int(input("Please enter a small positive integer: "))
    print('You entered', val)
    [][2] = 5   # Try to assign to a nonexistent index of the empty list
except ValueError:
    print('Input not accepted')
```

Output

```
Please enter a small positive integer: 5
You entered 5
Traceback (most recent call last):
  File "enterintexcept.py", line 4, in <module>
    [][2] = 5   # Try to assign to a nonexistent index of the empty list
IndexError: list assignment index out of range
```

Handling exceptions

- Handle multiple exceptions

Codes

```
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
    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)
```

Handling exceptions

- Handle multiple exceptions: Output

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

```
Beginning of loop iteration 5
List index is out of range
End of loop iteration 5
Beginning of loop iteration 6
Division by zero not allowed
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
List index is out of range
End of loop iteration 8
Beginning of loop iteration 9
Division by zero not allowed
End of loop iteration 9
```

Handling exceptions

- Multiple types of exceptions in a single *except* block

```
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
    except (ValueError, ZeroDivisionError):
        print('Problem with integer detected')
    except IndexError:
        print('List index is out of range')

    print('End of loop iteration', i)
```

Handling exceptions

- What if another type of exception (not one of the three types) occurs?

Codes

```
import random

for i in range(10):    # Loop 10 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')

    print('End of loop iteration', i)
```

Handling exceptions

- What if another type of exception (not one of the three types) occurs?

Output

```
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
Traceback (most recent call last):
  File "missedexception.py", line 12, in <module>
    print({}[1])  # Try to use a nonexistent key to get an item from a dictionary
KeyError: 1
```

- Do we need to list all the types of exceptions to to handle all possible exceptions that can arise?

Handling exceptions

- The type *Exception* matches any exception type that a programmer would reasonably want to catch.

Codes

```
import random

for i in range(10):    # Loop 10 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
```

Codes

```
    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)
```

Output

```
List index is out of range
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
Cannot convert integer
End of loop iteration 3
```

Output

```
Beginning of loop iteration 4
This program has encountered a problem
End of loop iteration 4
Beginning of loop iteration 5
Cannot convert integer
End of loop iteration 5
Beginning of loop iteration 6
Cannot convert integer
End of loop iteration 6
Beginning of loop iteration 7
Cannot convert integer
End of loop iteration 7
Beginning of loop iteration 8
Division by zero not allowed
End of loop iteration 8
Beginning of loop iteration 9
Division by zero not allowed
End of loop iteration 9
```

Handling exceptions

- We can inspect the object that an *except* block catches using the *as* keyword.

Codes

```
import random

for i in range(10):    # Loop 10 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
```

Codes

```
    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)
```

Output

```
Beginning of loop iteration 0
Problem with division  ==> <class 'ZeroDivisionError'> division by zero
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
```

Output

```
Beginning of loop iteration 4
Problem with division  ==> <class 'ZeroDivisionError'> division by zero
End of loop iteration 4
Beginning of loop iteration 5
Problem with list      ==> <class 'IndexError'> list assignment index out of range
End of loop iteration 5
Beginning of loop iteration 6
Problem with division  ==> <class 'ZeroDivisionError'> division by zero
End of loop iteration 6
Beginning of loop iteration 7
Problem with value     ==> <class 'ValueError'> invalid literal for int() with base 10: 'Fr
End of loop iteration 7
Beginning of loop iteration 8
Problem with value     ==> <class 'ValueError'> invalid literal for int() with base 10: 'Fr
End of loop iteration 8
Beginning of loop iteration 9
Problem with division  ==> <class 'ZeroDivisionError'> division by zero
End of loop iteration 9
```

Outline

- Handling exceptions
- **Exception handling scope**
- Raising exceptions
- The *else* and *finally* blocks

Exception handling scope

- Handling exceptions in functions

Codes

```
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


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
```

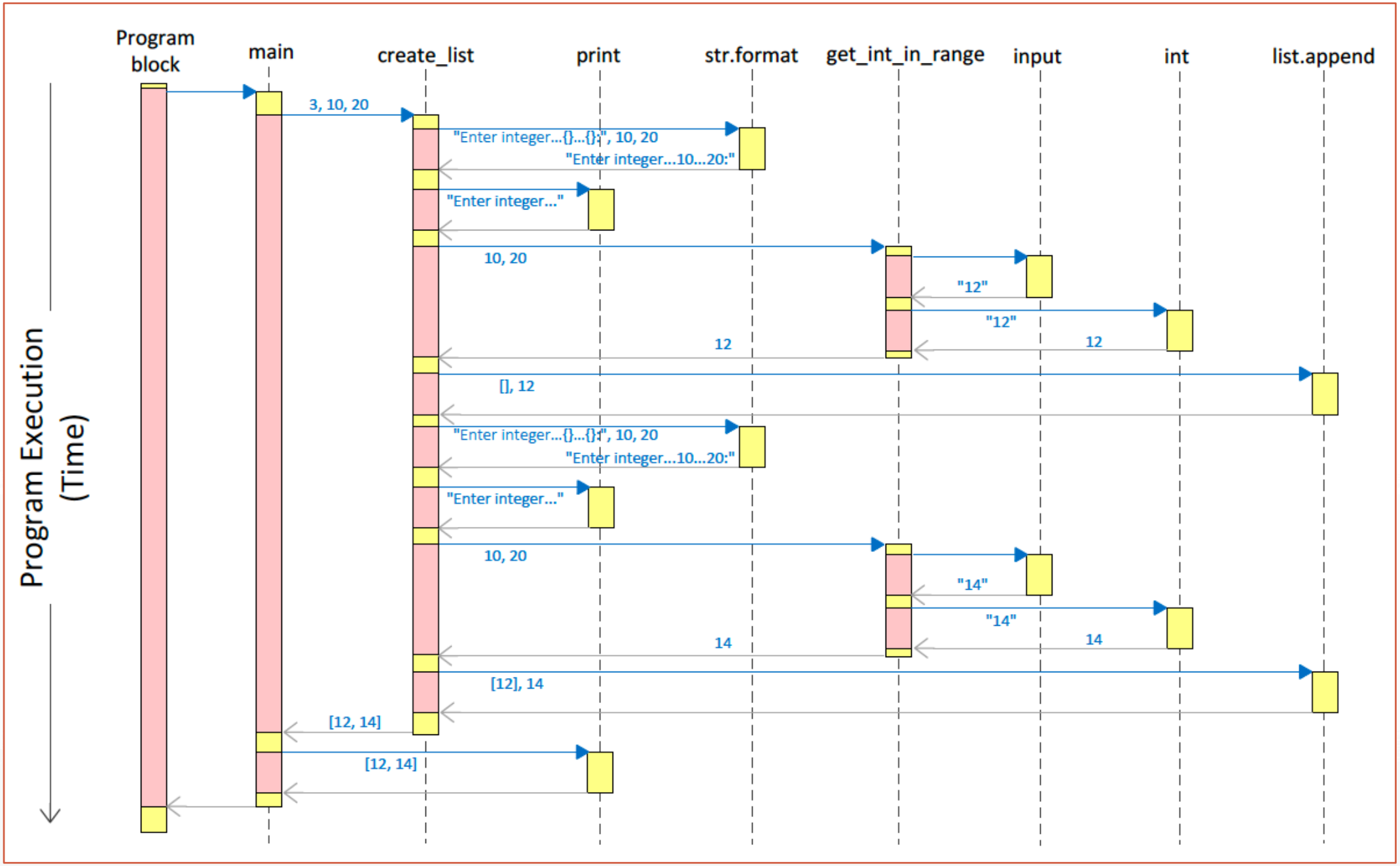
Codes

```
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
```

Output

```
Enter integer in the range 10...20: 12
Enter integer in the range 10...20: 14
[12, 14]
```

Diagram of function call



Exception handling scope

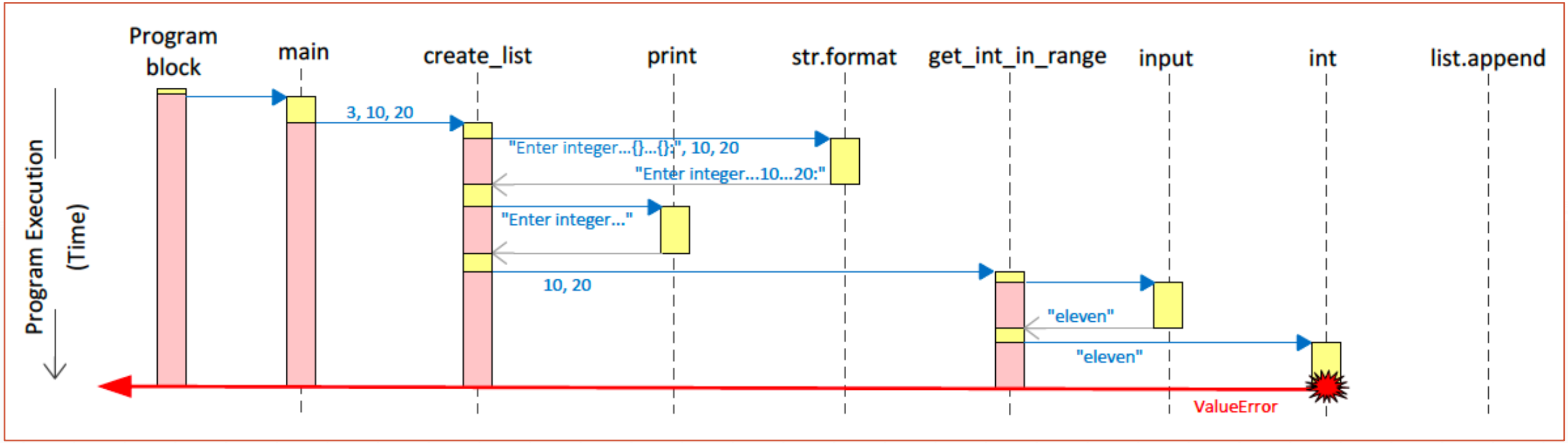
- How about the user type *eleven*?
  - When an exception occurs, it will propagate back up the call chain. If the propagation progresses all the way back to the program block level and along the way encounters no exception handler with an *except* block of its type, the program will stop execution.

Output

```
Enter integer in the range 10...20: eleven
Traceback (most recent call last):
  File "makeintegerlist.py", line 27, in <module>
    main()    # Invoke main
  File "makeintegerlist.py", line 23, in main
    lst = create_list(3, 10, 20)
  File "makeintegerlist.py", line 16, in create_list
    result.append(get_int_in_range(min, max))
  File "makeintegerlist.py", line 4, in get_int_in_range
    val = int(input())      # Can raise a ValueError
ValueError: invalid literal for int() with base 10: 'eleven'
```

Exception handling scope

- Diagram of function call



Exception handling scope

- Handle the exception in the function of *create_list*

Codes

```
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


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 = []
    try:
        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
    except ValueError:
        print('Disallowed user entry interrupted list creation')
    return result
```

Codes

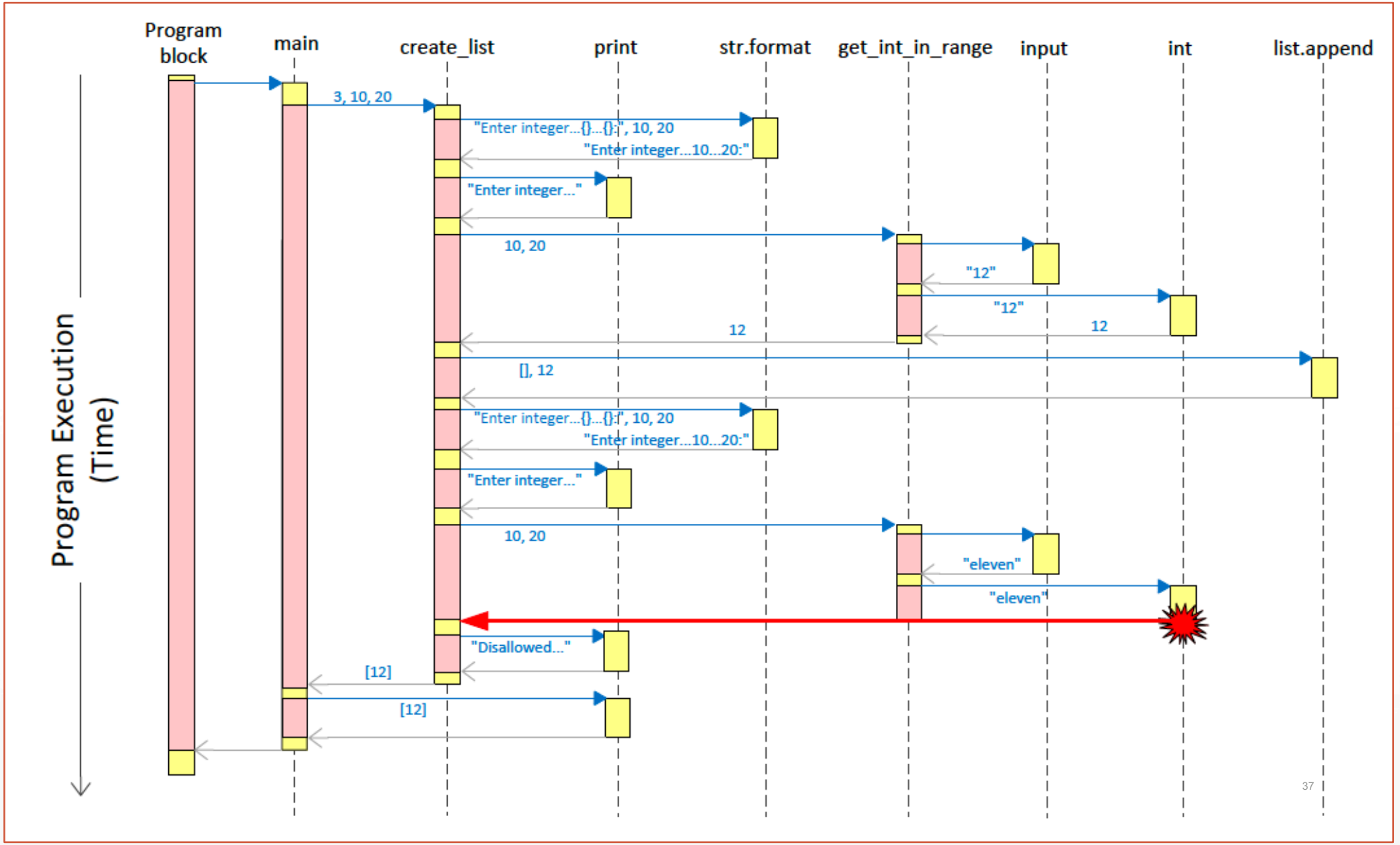
```
def main():
    """  Create a list of two elements supplied by the user,
         each element in the range 10...20 """
    lst = create_list(2, 10, 20)
    print(lst)


if __name__ == '__main__':
    main()    # Invoke main
```

Output

```
Enter integer in the range 10...20: 12
Enter integer in the range 10...20: eleven
Disallowed user entry interrupted list creation
[12]
```

Exception handling scope

Diagram of function call



Exception handling scope

- Handle the exception in the function of *get_int_in_range* instead of *create_list*

Codes

```
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

- Handle the exception in the function of *get_int_in_range* instead of *create_list*

Output

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

- The handler is close to the source of the exception and can utilize local information to address the exception.

Outline

- Handling exceptions
- Exception handling scope
- **Raising exceptions**
- The *else* and *finally* blocks

Raising exceptions

- The *raise* statement allows the programmer to force a specified exception to occur.
- The following statement produces an exception.

```
raise ValueError()
```

The argument should be an exception instance or class.

- The class constructor for most exception objects accepts a string parameter that provides additional information to handlers:

```
raise ValueError('45')
```

Raising exceptions

- Examples

```
>>> raise ValueError()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError
>>> raise ValueError('45')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 45
>>> raise ValueError('This is a value error')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: This is a value error
```

Raising exceptions

- Examples

Codes

```
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')
```

Raising exceptions

- Examples

Output

```
Please enter a nonnegative integer:3
You entered 3
Please enter a nonnegative integer:-3
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 we only need to determine whether an exception was raised but don't intend to handle it, we can use the raise statement 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 <module>
NameError: HiThere
```

Raising exceptions

- Example

Codes

```
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:
```

Codes

```
        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


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
```

Codes

```
    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


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()
```

Raising exceptions

- Example

Output

```
4
3
6 is a not an acceptable element
Traceback (most recent call last):
  File "reraise.py", line 48, in <module>
    main()
  File "reraise.py", line 44, in main
    print(count_elements([2, 4, '6', 8, 'x', 7], is_prime))
  File "reraise.py", line 33, in count_elements
    if predicate(x):
  File "reraise.py", line 9, in is_prime
    if n < 2 or n % 2 == 0:         # Handle simple cases immediately
TypeError: unorderable types: str() < int()
```

Raising exceptions

- We can wrap the function calling code with a *try* statement:

Codes

```
#  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')
```

Raising exceptions

- We can wrap the function calling code with a *try* statement:

Codes

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

- 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.

Outline

- Handling exceptions
- Exception handling scope
- Raising exceptions
- **The *else* and *finally* block**

The *else* and *finally* blocks

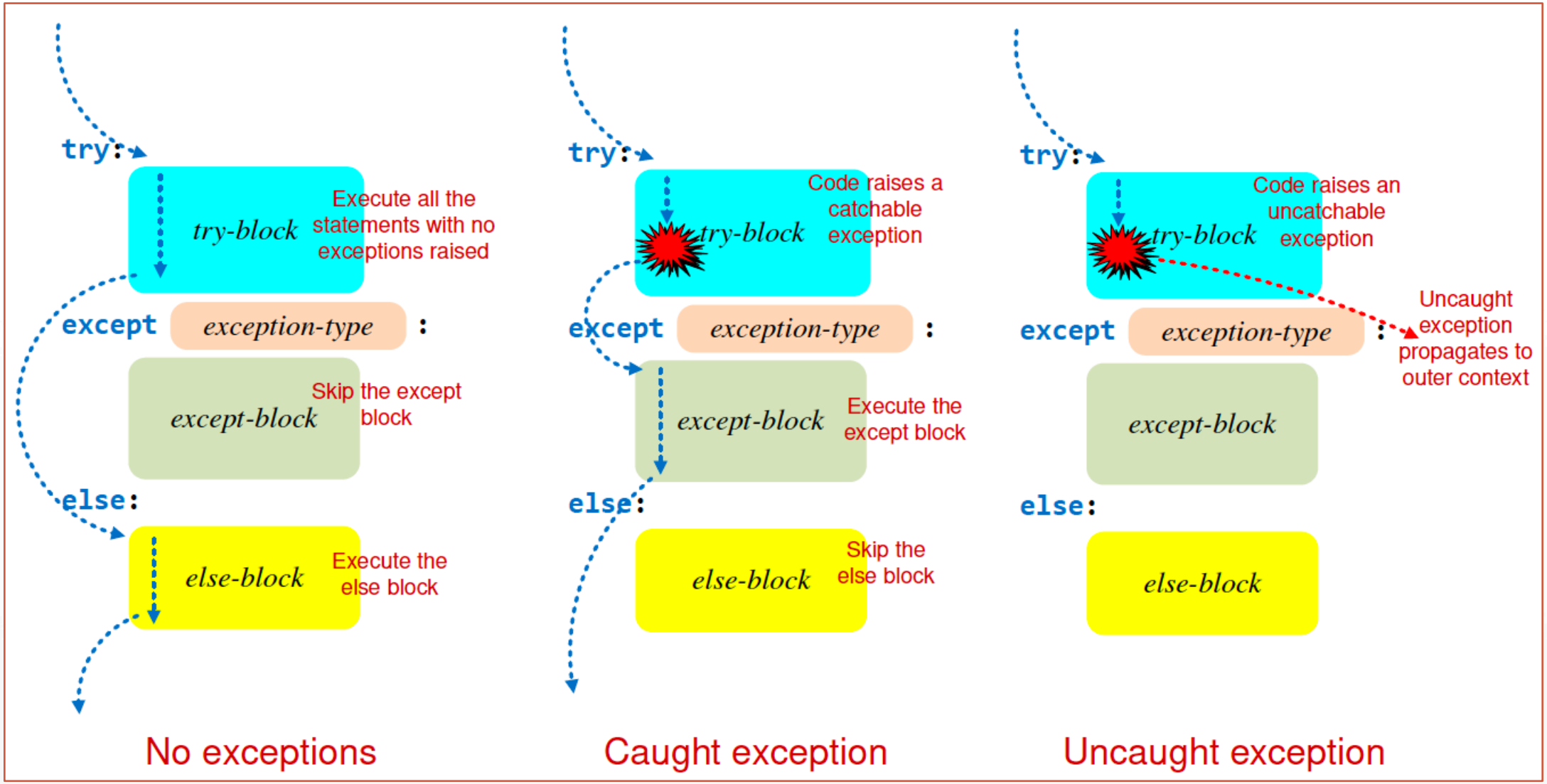
- The *try* statement has an optional *else* block, which, when present, must appear after all *except* blocks.
- If the code within the *try* block does not produce an exception, no *except* blocks trigger, and the program's execution continues with code in the *else* block.
- It is useful for code that must be executed if the try block does not raise an exception.

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

https://docs.python.org/3/tutorial/errors.html#handling-exceptions

The *else* and *finally* blocks

- The *else* block



The *else* and *finally* blocks

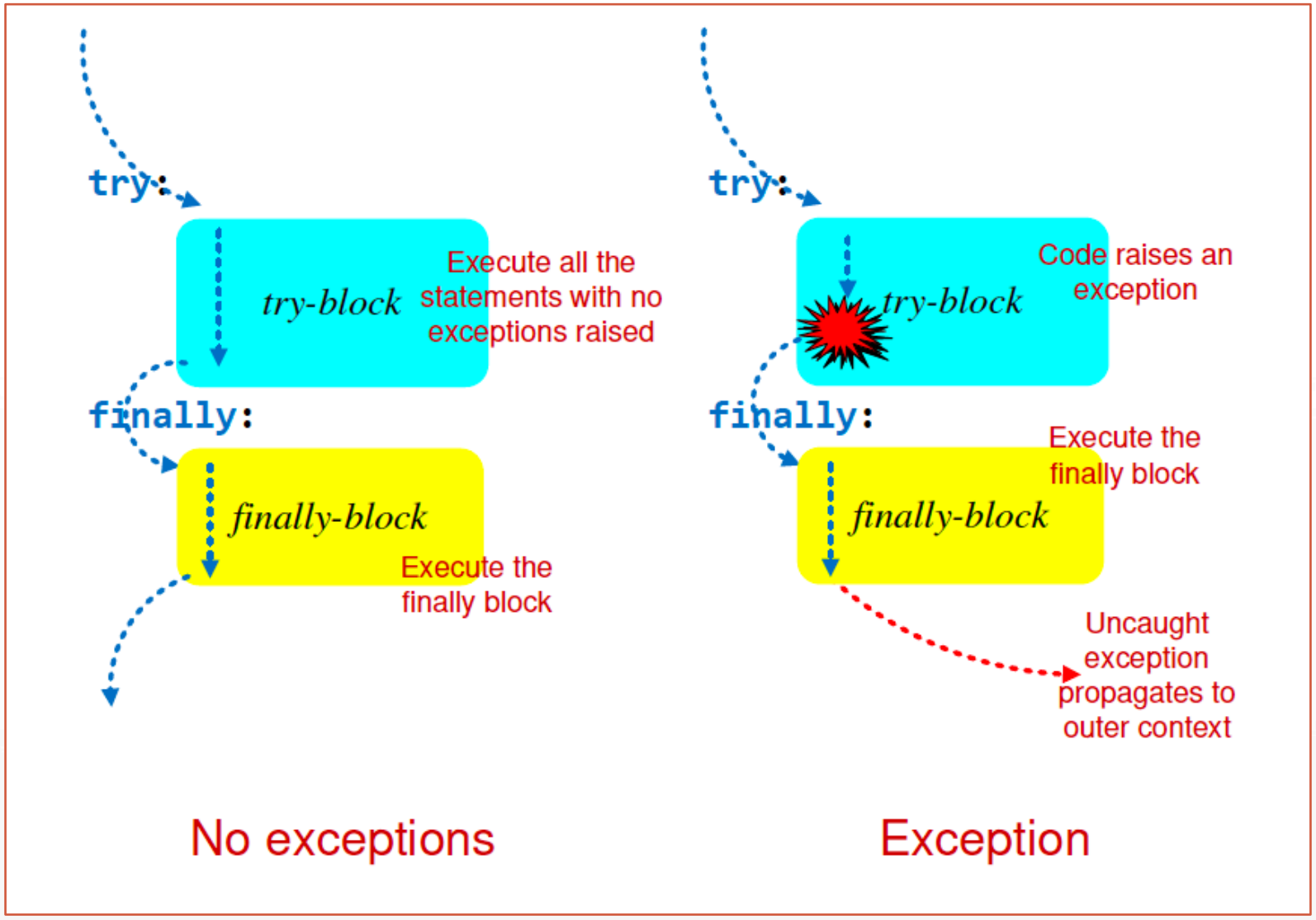
- The *try* statement has another optional *finally* block which is intended to define clean-up actions that must be executed under all circumstances.
- If a *finally* block is present, the *finally* block will execute as the last task before the *try* statement completes.
- The *finally* block runs whether or not the *try* statement produces an exception.

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

https://docs.python.org/3/tutorial/errors.html#handling-exceptions

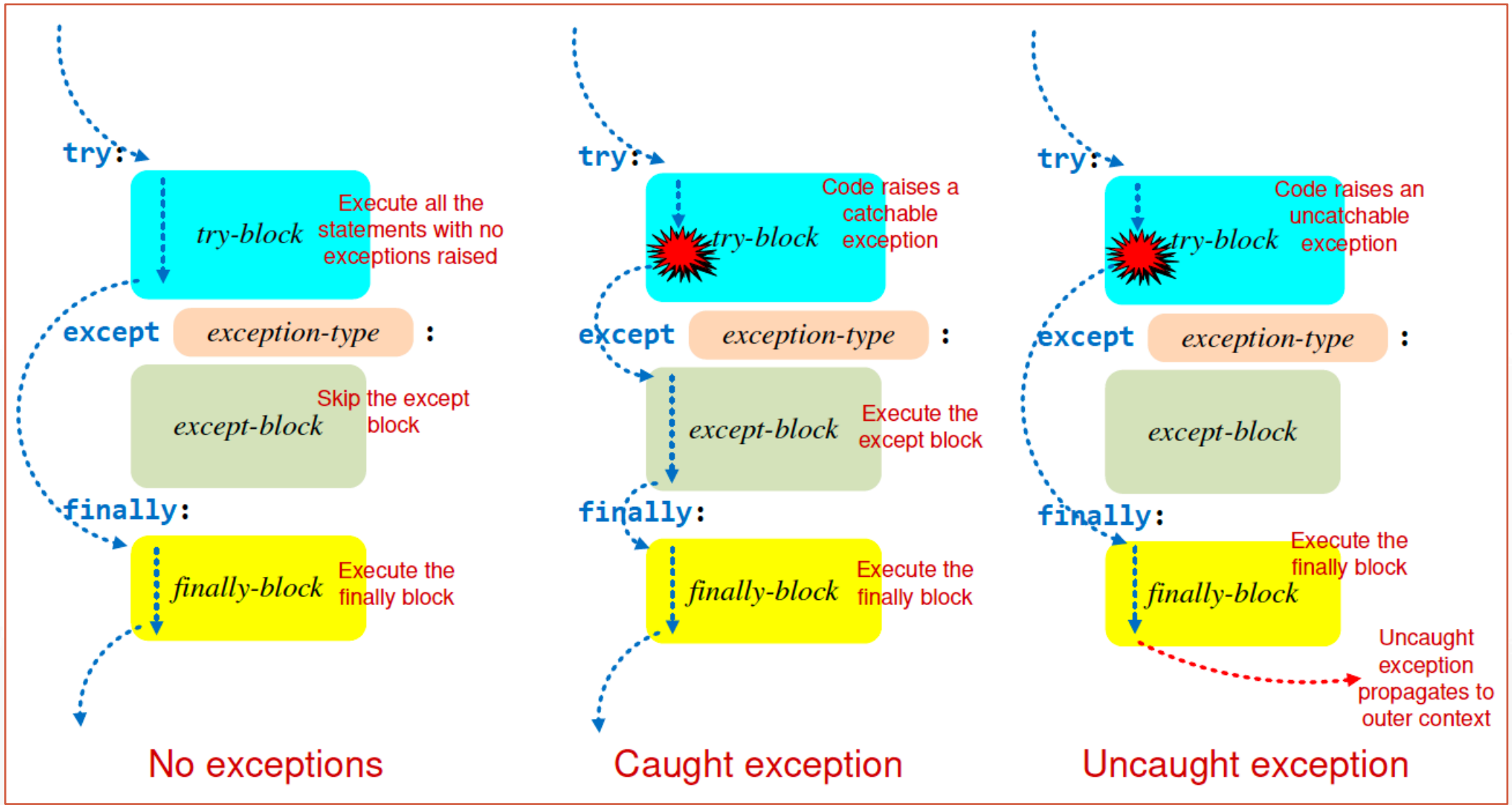
The *else* and *finally* blocks

- The *finally* block (no *except* block)



The *else* and *finally* blocks

- The *finally* block (with *except* block)



The *else* and *finally* blocks

- Example: the *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)
```

References

- *Think Python: How to Think Like a Computer Scientist*, 2nd edition, 2015, by Allen Downey
- *Python Cookbook*, 3rd edition, by David Beazley and Brian K. Jones, 2013
- *Python for Data Analysis*, 2nd edition, by Wes McKinney, 2018
- *Fundamentals of Python Programming,* by Richard L. Halterman