Computational Thinking

Lecture 12: Exception & Debugging

University of Engineering and Technology

Vietnam National University

Outline

- Syntax error
- Exception
- Handling exception
  - With if statement
  - With try-except block
  - Unhandled exception
  - Clean-up action
- Raise exception
- Assertion
- Debugging

Syntax Error & Exception

Syntax error

- Occurs when the code violates Python's grammatical rules
- Detected by the parser, before the program starts executing

```
while True
    print("Hello world")
```

```
  File "/tmp/ipython-input-2176175059.py", line 1
    while True
               ^
SyntaxError: expected ':'
```

Syntax error

- Occurs when the code violates Python's grammatical rules
- Detected by the parser, before executing

A colon (":") is missing, the code cannot be executed

```
while True
    print("Hello world")
```

```
  File "/tmp/ipython-input-2176175059.py", line 1
    while True
               ^
SyntaxError: expected ':'
```

How functions finish executing?

If a function is syntactically correct, how does it finish executing?

How functions finish executing

Return a value

- return statement: ends the function normally and sends back a value
- If no return is given, Python automatically returns None

Raise an exception

- Function stops before completing all its code
- Execution ends abnormally due to an exception is raised.
  - Some languages use the term “throw”

Exception

- Exception: errors detected during program execution
- They interrupt the normal flow of the program
- Examples: `ValueError`, `ZeroDivisionError`, `TypeError`, `FileNotFoundError`

Varieties of exceptions

- Python provides many built-in exceptions that help identify different kinds of runtime errors.
  - `ZeroDivisionError`: Division by zero
  - `IndexError`: List index out of range
  - `ValueError`: Invalid value for an operation
  - `FileNotFoundError`: File not found on disk
  - `TypeError`: Operation on incompatible data types
  - More details

Example of exception

```
def divide(a, b):
    """
    Divide a by b and return the result.
    Raises an exception if b is zero.
    """
    return a / b

res = divide(10, 0)
print(res)
```

Example of exception

```
def divide(a, b):
    """
    Divide a by b and return the result.
    Raises an exception if b is zero.
    """
    return a / b

res = divide(10, 0)
print(res)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
/tmp/ipython-input-1061594283.py in <cell line: 0>()
      6   return a / b
      7
----> 8 res = divide(10, 0)
      9 print(res)

/tmp/ipython-input-1061594283.py in divide(a, b)
      4   Raises an exception if b is zero.
      5   """
----> 6   return a / b
      7
      8 res = divide(10, 0)

ZeroDivisionError: division by zero
```

Example of exception

```
import random
the_num = random.randint(1, 100)
guess = input('What number am I thinking of?')
num = int(guess)
if num == the_num:
      print('Correct!')
else:
      print('Sorry, that was not it.')
```

What kinds of problems could happen when this code runs?

Example of exception

```
import random
the_num = random.randint(1, 100)
guess = input('What number am I thinking of?')
num = int(guess)
if num == the_num:
        print('Correct!’)
else:
        print('Sorry, that was not it.')
```

```
What number am I thinking of?abc
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipython-input-2309708468.py in <cell line: 0>()
      2 the_num = random.randint(1, 100)
      3 guess = input('What number am I thinking of?')
----> 4 num = int(guess)
      5 if num == the_num:
      6   print('Correct!')

ValueError: invalid literal for int() with base 10: 'abc'
```

Example of exception

```
import random
the_num = random.randint(1, 100)
guess = input('What number
num = int(guess)
if num == the_num:
        print('Correct!')
else:
        print('Sorry, that was not it.')
```

`int(guess)` fails on non-numeric input (e.g., "abc")
→ `ValueError`

```
What number am I thinking of?abc
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipython-input-2309708468.py in <cell line: 0>()
      2 the_num = random.randint(1, 100)
      3 guess = input('What number am I thinking of?')
----> 4 num = int(guess)
      5 if num == the_num:
      6   print('Correct!')

ValueError: invalid literal for int() with base 10: 'abc'
```

Handling Exception

Exception handling with if statement

```
def divide(a, b):
    if b == 0:
        print("Error: Cannot divide by zero.")
        return None
    return a / b

res = divide(10, 0)
print(res)
```

Exception handling with if statement

```
def divide(a, b):
    if b == 0:
        print("Error: Cannot divide by zero.")
        return None
    return a / b

res = divide(10,
print(res)
```

The `if` statement checks whether `b` is zero and prevents the program from executing the invalid operation `a / b`.

Exception handling with if statement

```
import random
the_num = random.randint(1, 100)
guess = input('What number am I thinking of?')
if is int(guess):
    num = int(guess)
    if num == the_num:
        print('Correct!')
    else:
        print('Sorry, that was not it.')
else:
    print('Sorry, you did not enter a number')
```

That would be nice, but there is no built-in function `is_int()` in Python

Exception handling with try statement

```
import random
the_num = random.randint(1, 100)
guess = input('What number am I thinking of?')
try:
    num = int(guess)
    if num == the_num:
        print('Correct!’)
    else:
        print('Sorry, that was not it.’)
except:
    print('Sorry, you did not enter a number')
```

Exception handling with try statement

```
import random
the_num = random.randint(1, 100)
guess = input('What number am I thinking of?')
try:
    num = int(guess)
    if num == the_num:
        print('Correct!’)
    else:
        print('Sorry, that was not it.’)
except:
    print('Sorry, you did not enter a number')
```

try executing this code

Exception handling with try statement

```
import random
the_num = random.randint(1, 100)
guess = input('What number am I thinking of?')
try:
    num = int(guess)
    if num == the_num:
        print('Correct!’)
    else:
        print('Sorry, that was not it.’)
except:
    print('Sorry, you did not enter a number')
```

if an error occurs, give up and fail over to this code

iClicker Question

```
def get_item(lst, idx):
try:
    return lst[idx]
except:
    return 'Out of bounds'
print('Will this run?')

print(get_item([1, 2, 3], 5))
```

What is the output?
A. Prints None
B. Prints ‘Out of bounds’
C. Prints ‘Will this run?’ then ‘Out of bounds’
D. Prints ‘Out of bounds’ then ‘Will this run?’
E. Prints ‘IndexError’

Try statement

- Syntax

```
try:
    <Statements>
except:
    <statements>
```

Try statement

```
try:
    guess = input("Input an int: ")
    num = int(guess)
    print('The number is ', num)
except:
    print("Error! it's not an int")
print("Done.")
```

Output:

```
Input an int: 5
The number is  5
Done.
```

Explanation:

input is **5**, which is **valid**
no exception
the except block is **ignored**

Try statement

```
try:
    guess = input("Input an int: ")
    num = int(guess)
    print('The number is ', num)
except:
    print("Error! it's not an int")
print("Done.")
```

Output:

```
Input an int: abc
Error! it's not an int
Done.
```

Explanation:

input **abc**, which is **invalid**, **an exception occur**
the remaining code in try is **ignored**, the except block is executed

Try statement – named exceptions

Syntax

```
try:
    <Statements>
except <exn-name>:
    <statements>
except <exn-name>:
    <statements>
…
except:
    <statements>
```

Example

```
try:
    guess = input('?')
    num = int(guess)
except ValueError:
    print('Not an int' )
except TypeError:
    print()
except:
    print('Sorry')
```

When an exception is raised in `try`:

- Run the matching `except` block.
- Otherwise, run the default `except` block.

Try statement – no default

Syntax

```
try:
    <Statements>
except <exn-name>:
    <statements>
…
except <exn-name>:
    <statements>
```

Example

```
try:
    guess = input('?')
    num = int(guess)
except ValueError:
    print('Not an int' )
except TypeError:
    print()
```

- If no matching or default except exists, the error is unhandled.
- The program stops when the exception propagates.

Unhandled exception

- Sometimes exceptions go **unhandled**
  - This can happen when:
    - There is no `try` statement
    - There is no `except` block for the corresponding except type
    - There is no default `except` block
- When unhandled, the exception is passed up to the next frame on the call stack
  - It may be caught and handled there
  - If not, it continues propagating upward
  - If it's never handled, program execution ends with an error

  → the program crashes

Example: Unhandled exceptions are raised up

```
from typing import IO

def str_to_int(s):
    return int(s)

def func1():
    try:
        num = str_to_int('apple’)
        print(num + 1)
    except IOError:
        print('Not a number')

func1()
```

Example: Unhandled exceptions are raised up

This is the output of executing the program in the previous s

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipython-input-2873279497.py in <cell line: 0>()
     10     print('Not a number')
     11
---> 12 func1()

 1 frames
/tmp/ipython-input-2873279497.py in func1()
      5 def func1():
      6   try:
----> 7     num = str_to_int('apple')
      8     print(num + 1)
      9   except IOError:

/tmp/ipython-input-2873279497.py in str_to_int(s)
      1 from typing import IO
      2 def str_to_int(s):
----> 3   return int(s)
      4
      5 def func1():

ValueError: invalid literal for int() with base 10: 'apple'
```

Example: Unhandled exceptions are raised up

This is the output of executing the program in the previous slide

Example: Unhandled exceptions are raised up

This is the output of executing the program in the previous slide

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipython-input-2873279497.py in <cell line: 0>()
     10     print('Not a number')
     11
---> 12 func1()

/tmp/ipython-input-287327949
      5 def func1():
      6   try:
----> 7     num = str_to_int('apple')
      8     print(num + 1)
      9   except IOError:

/tmp/ipython-input-2873279497.py in str_to_int(s)
      1 from typing import IO
      2 def str_to_int(s):
----> 3   return int(s)
      4
      5 def func1():

ValueError: invalid literal for int() with base 10: 'apple'
```

(2) Enter `try` block

Example: Unhandled exceptions are raised up

This is the output of executing the program in the previous slide

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipython-input-2873279497.py in <cell line: 0>()
     10     print('Not a number')
     11
---> 12 func1()

/tmp/ipython-input-2873279497.py
      5 def func1():
      6   try:
----> 7     num = str_to_int('ap
      8     print(num + 1)
      9   except IOError:

/tmp/ipython-input-2873279497.py in      int(s)
      1 from typing import IO
      2 def str_to_int(s):
----> 3   return int(s)
      4
      5 def func1():

ValueError: invalid literal for int() with base 10: 'apple'
```

(3) A `ValueError` exception is raised from `str_to_int`(s) but it is not handled and passed to its caller `func1()`

Example: Unhandled exceptions are raised up

This is the output of executing the program in the previous slide

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipython-input-2873279497.py in <cell line: 0>()
     10     print('Not a number')
     11
---> 12 func1()

 ------------------------------------------------------------

/tmp/ipython-input-2873279497.py in
      5 def func1():
      6   try:
----> 7     num = str_to_int('apple')
      8     print(num + 1)
      9   except IOError:

/tmp/ipython-input-2873279497.py in str_to_int(s)
      1 from typing import IO
      2 def str_to_int(s):
----> 3   return int(s)
      4
      5 def func1():

ValueError: invalid literal for int() with base 10: 'apple'
```

(4) `func1()` does not handle `ValueError` exception → the program crashed

Clean-up actions: finally

Syntax

```
try:
    <Statements>
except <exn-name>:
    <statements>
…
except:
    <statements>
finally:
    <statements>
```

Clean-up actions: finally

```
def divide(a, b):
    try:
        result = a/b;
        print("Result is ", result)
    except ZeroDivisionError:
        print("Error: Cannot divide by zero.")
    finally:
         print("Cleaning up and completing")

res = divide(10, 0)
print(res)
```

Output

```
Error: Cannot divide by zero.
Cleaning up and completing the execution
```

Clean-up actions: finally

- The `finally` block always runs:
  - whether or not the `try` block produces any exceptions
  - Whether or not the exception is caught and handled
- Use the `finally` block to release resources
  - Close files
  - Disconnect from networks
  - Release memory or locks

Raise Exception

Where do exceptions come from?

- Exception are raised by Python when something goes wrong (invalid input, file not found, etc.)

e.g. `num = int('apple')` → `ValueError`

- Who raised exceptions?
  - In this example, the bult-in function `int()`
  - We can raise exceptions in our defined functions

Why raise exceptions?

- To indicate invalid inputs or logical errors
- To stop execution when an issue occurs
- To enforce clear and intentional error handling

Example: Raise exception

```
def check_password(pwd):
    if len(pwd) < 6:
        raise ValueError("Password must be at
                          least 6 characters long.")
    if pwd.isalpha() or pwd.isdigit():
        raise ValueError("Password must contain
                  both letters and numbers.")
    print("Password is valid!")

try:
    check_password("abc12")
except ValueError as e:
    print("Invalid password:", e)
```

Raise exception

- **Syntax**

```
raise <Exception>
```

- **Example**

```
raise ValueError("not a number")
raise TypeError("invalid type")
raise Exception("msg")
```

Assertion

Assertion

- An `assertion` is a specialized tool for detect and handle errors early
  - Raise an exception if a condition is not met
- Purpose:
  - Detect programmer errors
  - Debug logic during development
- Common uses:
  - Testing: `assert _equals()`
  - Debugging: `assert` statements

Example: A programmer error

```
def greet(name):
    print("Hello," + name)

for n in ["Roy", 6]:
    greet(n)
```

```
Hello,Roy
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipython-input-2946180275.py in <cell line: 0>()
      3
      4 for n in ["Roy", 6]:
----> 5   greet(n)

/tmp/ipython-input-2946180275.py in greet(name)
      1 def greet(name):
----> 2   print("Hello," + name)
      3
      4 for n in ["Roy", 6]:
      5   greet(n)

TypeError: can only concatenate str (not "int") to str
```

Example: A programmer error

```
def greet(name):
    print("Hello," + name)

for n in ["Roy", 6]:
```

This error is the programmer's fault. They violated the precondition because 6 is not a string. But it requires some thought to figure that out from the error message. As programs get more complicated, the amount of thought grows. Let's make it easier to find the error.

```
----> 2   print("Hello," + name)
      3
      4 for n in ["Roy", 6]:
      5   greet(n)

TypeError: can only concatenate str (not "int") to str
```

Using an assert to check the precondition

```
def greet(name):
    assert isinstance(name, str), \
        "name must be a string but " \
        + str(name) + " is not"
    print("Hello," + name)

for n in ["Roy", 6]:
    greet(n)
```

Using an assert to check the precondition

```
Hello,Roy
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
/tmp/ipython-input-3073616038.py in <cell line: 0>()
      5
      6 for n i
----> 7   greet

/tmp/ipython-in
      1 def gre
----> 2   assert isinstanc                    \
      3          "name must b          , but " + str(name) + " is not"
      4   print("Hello," + nam
      5

AssertionError: name must be a string but 6 is not
```

That's a more helpful error message for the programmer as they debug their code

Assertions are exceptions

- `AssertionError` is a type of exception
- `assert` statement is a specialized way of raising it
- Assertions can be disabled
  - Run python with a flag: `python -O script.py`
  - This executes the script but ignores all `assert` statements
  - Real world production code might do that to improve performance
- Use assertions to detect incorrectness, not to guarantee correctness
- Use of assertions for testing and debugging

Debugging

Bugs

- **Bug**: a mistake or flaw in program that causes unexpected behavior
  - Often due to wrong logic, incorrect implementation, wrong assumption, etc.
- Debugging: the process of identifying, isolating, and fixing bugs

How to debug

- Don’t ask:
  - Why doesn’t my code do what I want it to do?
- Instead, ask:
  - What is my code doing?

How to debug

Two ways to inspect your code:

- 1. Step through your code, drawing execution diagrams or using Python tutor

- 2. Use print to display key information:
  - Intermediate variable values
  - Function inputs and outputs

Example: Using print to debug

```
def last_name_first(full_name):
    print('DEBUG: full_name =' + repr(full_name))
    space_index = full_name.index(' ')
    print('DEBUG: space_index =' + repr(space_index))
    first = full_name[:space_index]
    print('DEBUG: first =' + repr(first))
    last = full_name[space_index + 1:]
    print('DEBUG: last =' + repr(last))
    return_value = last + ", " + first
    print('DEBUG: return_value = ' + repr(return_value))
    return return_value

last_name_first("Toni Marrison")
```

Example: Using print to debug

```
DEBUG: full_name ='Toni Marrison'
DEBUG: space_index =4
DEBUG: first ='Toni'
DEBUG: last ='Marrison'
DEBUG: return_value = 'Marrison, Toni'
'Marrison, Toni'
```

Now we can see what the code is doing!
We have evidence for what line is “wrong”!

The `repr()` built-in function

- `repr()`: returns the canonical string representation of a value
  - a representation useful for programmers and debugging, not for end user

Using print to debug

- Quick and simple way to check what the code is doing
- Temporary debugging: useful for tracing variable values or program flow
- Not suitable for large projects: can causes messy for the outputs when the project grows
  - Use `logging` module instead for flexible, scalable debugging

Logging module

```
import logging

logging.basicConfig( level=logging.DEBUG,
format="%(levelname)s: %(message)s")

def last_name_first(full_name):
    logging.debug('full_name = %r', full_name)
    space_index = full_name.index(' ')
    logging.debug('space_index = %r', space_index)
    first = full_name[:space_index]
    logging.debug('first = %r', first)
    last = full_name[space_index + 1:]
    logging.debug('last = %r', last)
    return_value = last + ", " + first
    logging.debug('return_value = %r', return_value)
    return return_value
```

Logging module

- Structured and controllable system for recording program events
- Easily adjustable: enable or suppress output by setting the logging level (e.g., `logging.INFO`, `logging.ERROR`, `logging.DEBUG`)
- Rich formatting: include timestamp, log level, module names, etc.
- Suitable for production-scale: scalable and configurable

For more details:
https://docs.python.org/3/library/logging.html

Summary

- Exceptions interrupt normal program execution.
- Use `try-except` blocks to handle exceptions gracefully and keep the program running.
- Add a `finally` block to ensure cleanup actions always happen.
- Use `raise` to signal error intentionally when input or logic is invalid.
- Apply debugging techniques (e.g., `print`, `logging`) to inspect program behavior.