Lecture 10-2
Testing, Debugging and Exceptions

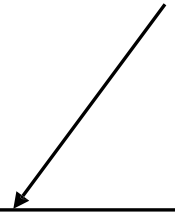
GNBF5010

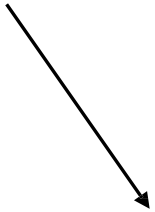
Instructor: Jessie Y. Huang

Testing & Debugging

DEFENSIVE PROGRAMMING

- Write **specifications** for functions
- **Modularize** programs
- Check **conditions** on inputs/outputs (assertions)





TESTING/VALIDATION

- **Compare** input/output pairs to specification
- "It's not working!"
- "How can I break my program?"

DEBUGGING

- **Study events** leading up to an error
- "Why is it not working?"
- "How can I fix my program?"

Set yourself up for easy testing and debugging

- From the start, design code to ease the testing and debugging
- Break program up into modules that can be tested and debugged individually
- Document constraints on modules
  - What do you expect the input to be?
  - What do you expect the output to be?
- Document assumptions behind code design

Types of program errors

- Syntax errors
  - Errors due to the violation of language rules.
  - Usually caught by Python
- Semantic errors:
  - Errors due to the use of meaningless statements, divide by 0 etc.
- Logic errors:
  - Errors due to the fact that the program doesn't fulfill the expectations. Wrong answers, incomplete results etc.

- **Testing** can help us avoid both **semantic** and **logic** errors

Test what?

- ***Correctness***. Are the results produced correct? What counts as correct? Do we know all the cases?
- ***Completeness***. Are all the potential cases covered: all user entries, all file formats, all data types, etc.
- ***Load***. Does the program respond well if it is presented with a heavy load? e.g. large data sets
- ***Resources***. Does the program use appropriate amounts of memory, CPU, network?
- ***Responsiveness***. Does the program respond in a reasonable amount of time. If there are time limits, does it meet those limits?

Debugging

- Steep learning curve
- Goal is to have a bug-free program
- tools
  - built-in to **IDLE** and **Anaconda**
  - **IDE**s like PyCharm, Visual Studio Code
  - Python Tutor
  - **print()** function
  - …

Use the print() function for debugging

- Good way to test hypothesis
- When to print
  - Start of a function
  - parameters
  - end of a function
- Use bisection method
  - put print halfway in code
  - decide where bug may be depending on values

Debug with error messages - Easy

- trying to access beyond the limits of a list
  `test = [1,2,3]` then `test[4]` → `IndexError`
- trying to convert an inappropriate type
  `int(test)` → `TypeError`
- referencing a non-existent variable
  `a` → `NameError`
- mixing data types without appropriate coercion → `TypeError`
- forgetting to close parenthesis, quotation, etc.

```
a = len([1,2,3]
print(a)
```
→ `SyntaxError`

Debug logic errors – Not easy

- Think before writing new code
- Draw pictures, take a break
- Explain the code to someone else

DON'T

- Write entire program
- Test entire program
- Debug entire program

DO

- Write a function
- Test the function, debug the function
- Write a function
- Test the function, debug the function
- *** Do integration testing ***

- Change code
- Remember where bug was
- Test code
- Forget where bug was or what change you made
- Panic

- Backup code
- Change code
- Write down potential bug in a comment
- Test code
- Compare new version with old version

Exceptions

How to deal with an exception?

- An **exception** is an **error** that occurs while a program is running, causing the program to abruptly **halt**.
- For instance, dividing by 0 raises the following exception:

```
Traceback (most recent call last):
  File "20.py", line 13, in <module>
    main()
  File "20.py", line 9, in main
    result = num1 / num2
ZeroDivisionError: division by zero
```

- One way to handle raised exceptions is an `if` statement:

```
num1 = 5
num2 = 0
if num2 != 0:
        result = num1 / num2
        print(num1, 'divided by', num2, 'is', result)
else:
        print('Cannot divide by 0.')
```

How to deal with an exception?

- The exception handler is a better way of handling exceptions

```
try:
  statement
  statement
  etc.
except ExceptionName:
  statement
  statement
  etc.
```

- Statements within the **try suite** have the potential of **raising** an exception.
- **ExceptionName** is optional

How try/except statement works?

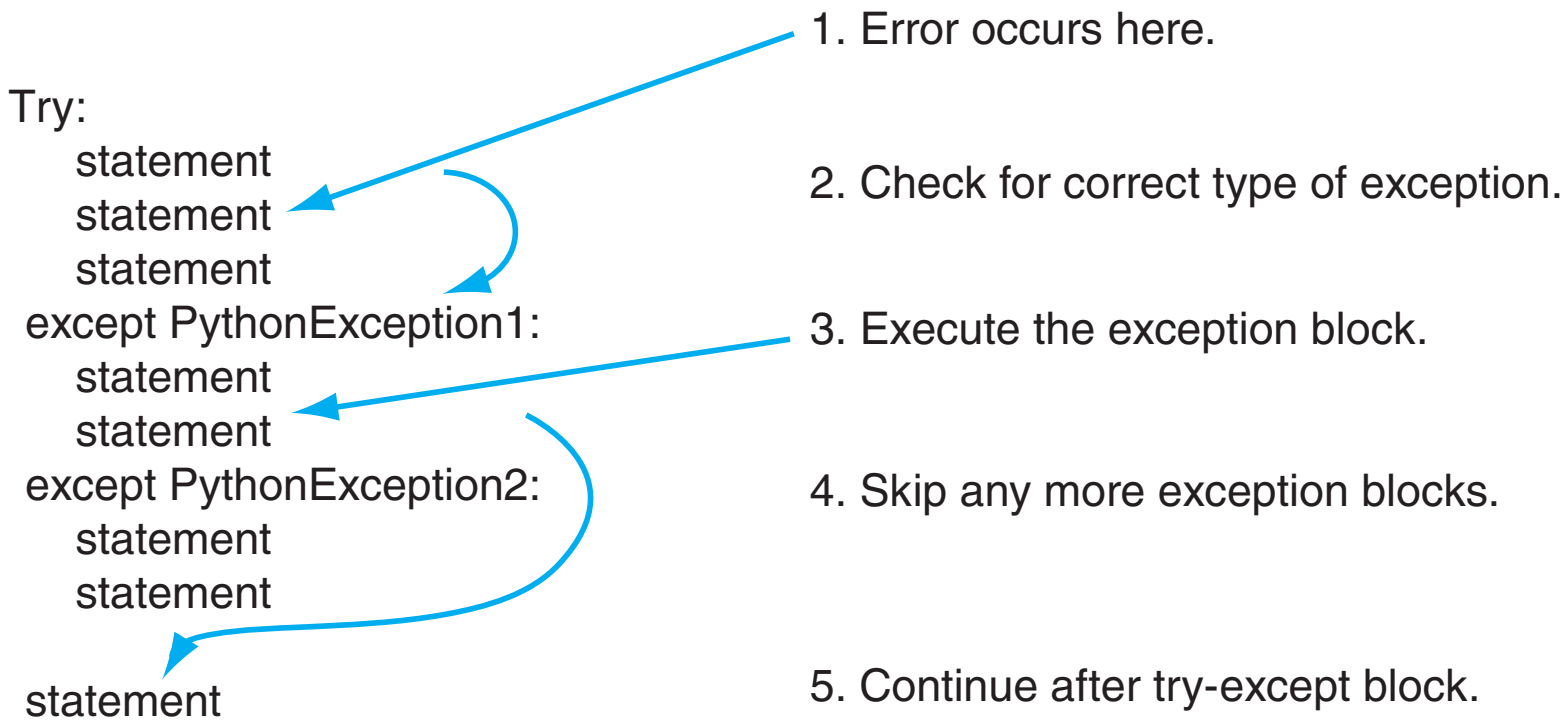


FIGURE 14.5 Exception flow.

How try/except statement works?

- First, the statements in the try suite begin to execute.
- Then:
  - If a statement within the try suite *raises* an **exception** specified by the **`ExceptionName`**, the control executes the corresponding exception block and continues to the next block after the try/except statement.
  - If not specified by `ExceptionName`, the program halts with a **traceback error message**.
  - If no statements in the try suite raise exceptions, the control goes to statements after the `try/except` statement.
- To catch all exceptions, simply **omit** the `ExceptionName`

```
try:
    print("Entering the try suite")
    dividend = float(input("Provide a dividend to divide:"))
    divisor = float(input("Provide a divisor to divide by:"))
    result = dividend/divisor
    print("{:2.2f} divided by {:2.2f} yields {:2.2f}".\
        format(dividend,divisor,result))
except ZeroDivisionError:
    print("Divide by 0 error")
except ValueError:
    print("Value error, could not convert to a float")

print("Continuing on with the rest of the program")
```

```
>>>
Entering the try suite
Provide a dividend to divide:10
Provide a divisor to divide by:2
10.00 divided by 2.00 yields 5.00
Continuing on with the rest of the program
>>> ================================ RESTART ================================
>>>
Entering the try suite
Provide a dividend to divide:10
Provide a divisor to divide by:a
Value error, could not convert to a float
Continuing on with the rest of the program
```

```
>>> ================================ RESTART ================================
>>>
Entering the try suite
Provide a dividend to divide:10
Provide a divisor to divide by:0
Divide by 0 error
Continuing on with the rest of the program
>>> ================================ RESTART ================================
>>>
Entering the try suite
Provide a dividend to divide:

Traceback (most recent call last):
  File "/Users/bill/book/v3.5/chapterExceptions/divide.py", line 3, in <module>
    dividend = float(input("Provide a dividend to divide:"))
KeyboardInterrupt
>>>
```

Handling exceptions

- Once you catch the exception, you need to handle it:
  - either perform an alternate action or
  - generate a customized error message and gracefully end the program

catch.py

```
import sys

try:
    fin = open('bad_file')
    for line in fin:
        print(line)
    fin.close()
except IOError:
    # exit the program if fails to open the file
    print('Cannot open the file. Program ends.')
    sys.exit(1)

# more code below
```

Python built-in exceptions

- The full list of built-in exceptions is regrettably big.
  - http://docs.python.org/library/exceptions.html#bltin-exceptions
- Some of the more important are:
  - `IndexError` sequence subscript out of range
  - `KeyError` dictionary key not found
  - `ZeroDivisionError` division by 0
  - `ValueError` value is inappropiate for the built-in function
  - `TypeError` function or operation using the wrong type
  - `IOError` input/output error, file handling

Use as to store exception in a variable

catch2.py

```
import sys

try:
    infile = open('myfile.txt', 'r')
except IOError as error:
    print("Can't open file:", str(error))
    sys.exit(1)

for line in infile:
    print(line, end='')
infile.close()

# More code below
```

Example

```
# Prompt for three values: input file, output file, search string.
# Search for the string in the input file, write results to the
# output file

import sys
def process_file(i_file, o_file, a_str):
    ''' if the a_str is in a line of i_file, add stars
        to the a_str in line, write it out with the
        line number to o_file '''
    line_count_int = 1
    for line_str in i_file:
        if a_str in line_str:
            new_line_str = line_str.replace(a_str, '***'+a_str)
            print('Line {}: {}'.format(line_count_int, new_line_str),\
                  file=o_file)
        line_count_int += 1

try:
    in_file_str = input("File to search:")
    in_file = open(in_file_str, 'r', encoding='utf_8')
except IOError:
    print('{} is a bad file name'.format(in_file_str))
    sys.exit()

out_file_str = input("File to write results to:")
out_file = open(out_file_str, 'w')
search_str = input("Search for what string:")
process_file(in_file, out_file, search_str)
in_file.close()
out_file.close()
```

Example (continued)

Results after searching for `"This"`:

inFile.txt	outFile.txt
This is a test	Line 1: ***This is a test
This is only a test	
Do not pass go	Line 2: ***This is only a test
Do not collect $200	

Philosophy of Exception Handling

Dealing with problems

Two ways to deal with exceptions

- ***LBYL***: Look Before you Leap
- ***EAFP***: Easier to Ask Forgiveness than Permission

Look Before You Leap

- It means before we execute a statement, we check all aspects to make sure it executes correctly:
  - if it requires a string, check that
  - if it requires a dictionary key, check that
- However, it tends to make code messy.
- The heart of the code (what you want it to do) may be hidden by all the checking.

Easier to Ask Forgiveness than Permission (EAFP)

- It means run any statement you want, no checking is required; afterwards, “clean up any messes” by catching errors that occur
- The `try` suite code reflects what you want to do and the `except` code reflects what you want to do on error.
  - Cleaner separation!

- Code Python programmers support the ***EAFP*** approach:
  - Run the code, let the `except` suites deal with the errors. Don’t check first.

```
# check whether int conversion will raise an error, two examples.
# Python Idioms, http://jaynes.colorado.edu/PythonIdioms.html

#LBYL, test for the problematic conditions
def test_lbyl (a_str):
    if not isinstance(a_str, str) or not a_str.isdigit:
        return None
    elif len(a_str) > 10:    #too many digits for int conversion
        return None
    else:
        return int(a_str)

#EAFP, just try it, clean up any mess with handlers
def test_eafp(a_str):
    try:
        return int(a_str)
    except (TypeError, ValueError, OverflowError): #int conversion failed
        return None
```

Reference

- Chapters 14 & 15, *The Practice of Computing Using Python* (Pearson)