HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Chapter 9-10:
Testing, debugging, exceptions, assertions

Contents

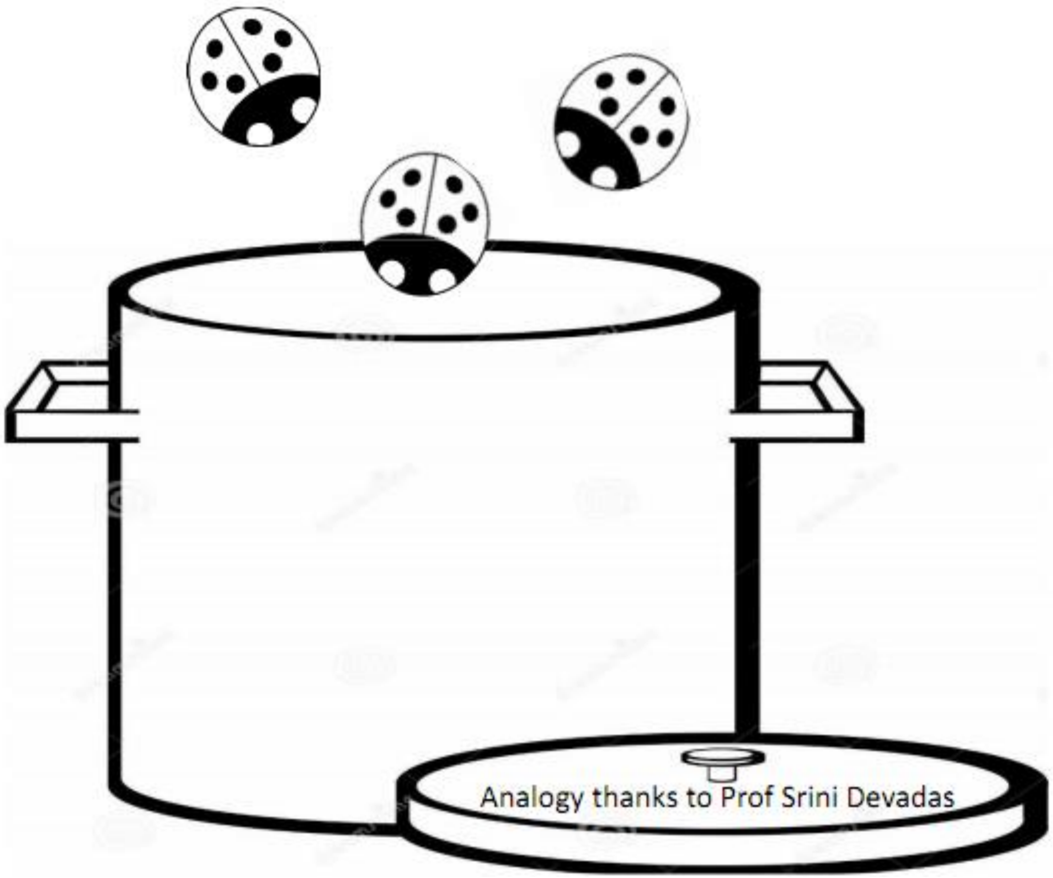
- Testing
- Debugging
- Exceptions
- Assertions

Testing

We aim for high quality – an analogy with soup

You are making soup but bugs keep falling in from the ceiling. What do you do?

- check soup for bugs
  - testing
- keep lid closed
  - defensive programming
- clean kitchen
  - eliminate source of bugs



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 this part
- 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

When are you ready to test?

- ensure **code runs**
  - remove syntax errors
  - remove static semantic errors
  - Python interpreter can usually find these for you
- have a **set of expected results**
  - an input set
  - for each input, the expected output

Classes of tests

- **Unit testing**
  - validate each piece of program
  - **testing each function** separately
- **Regression testing**
  - add test for bugs as you find them
  - **catch reintroduced** errors that were previously fixed
- **Integration testing**
  - does **overall program** work?
  - tend to rush to do this

Testing approaches

- **intuition** about natural boundaries to the problem

```
def is_bigger(x, y):
    """ Assumes x and y are ints
    Returns True if y is less than x, else False """
```

  - can you come up with some natural partitions?
- if no natural partitions, might do **random testing**
  - probability that code is correct increases with more tests
  - better options below
- **black box testing**
  - explore paths through specification
- **glass box testing**
  - explore paths through code

Black box testing

```
def sqrt(x, eps):
    """ Assumes x, eps floats, x >= 0, eps > 0
    Returns res such that x-eps <= res*res <= x+eps """
```

designed **without looking** at the code

- can be done by someone other than the implementer to avoid some implementer **biases**
- testing can be **reused** if implementation changes
- **paths** through specification
  - build test cases in different natural space partitions
  - also consider boundary conditions (empty lists, singleton list, large numbers, small numbers)

Black box testing

```python
def sqrt(x, eps):
    """ Assumes x, eps floats, x >= 0, eps > 0
    Returns res such that x-eps <= res*res <= x+eps """
```

CASE	x	eps
boundary	0	0.0001
perfect square	25	0.0001
less than 1	0.05	0.0001
irrational square root	2	0.0001
extremes	2	1.0/2.0**64.0
extremes	1.0/2.0**64.0	1.0/2.0**64.0
extremes	2.0**64.0	1.0/2.0**64.0
extremes	1.0/2.0**64.0	2.0**64.0
extremes	2.0**64.0	2.0**64.0

Glass box testing

- **use code** directly to guide design of test cases
- called **path-complete** if every potential path through code is tested at least once
- what are some **drawbacks** of this type of testing?
  - can go through loops arbitrarily many times
  - missing paths
- guidelines
  - branches
  - for loops
  - while loops

exercise all parts of a conditional

loop not entered
body of loop executed exactly once
body of loop executed more than once

same as for loops, cases that catch all ways to exit loop

Glass box testing

```python
def abs(x):
    """ Assumes x is an int
    Returns x if x>=0 and -x otherwise """
    if x < -1:
        return -x
    else:
        return x
```

- a path-complete test suite could **miss a bug**
- path-complete test suite: 2 and -2
- but abs(-1) incorrectly returns -1
- should still test boundary cases

Debugging

Debugging

- steep learning curve
- goal is to have a bug-free program
- tools
  - **built in** to IDLE and Anaconda
  - **Python Tutor**
  - **print**statement
  - use your brain, be **systematic** in your hunt

Print statements

- good way to **test hypothesis**
- when to print
  - enter function
  - parameters
  - function results
- use **bisection method**
  - put print halfway in code
  - decide where bug may be depending on values

Debugging steps

- **study** program code
  - don’t ask what is wrong
  - ask how did I get the unexpected result
  - is it part of a family?

- **scientific method**
  - study available data
  - form hypothesis
  - repeatable experiments
  - pick simplest input to test with

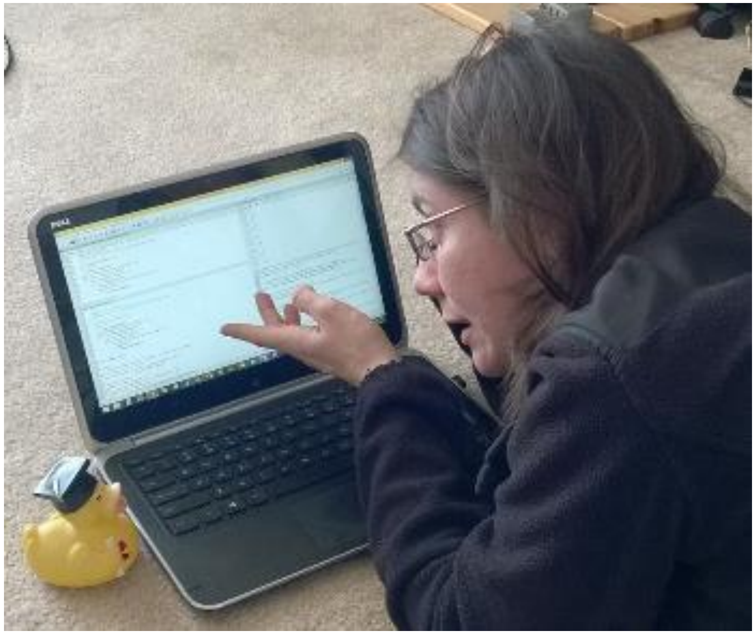
Error messages – easy

- 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
  `'3'/4` → `TypeError`
- forgetting to close parenthesis, quotation, etc.

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

Logic errors - hard

- **think** before writing new code
- **draw** pictures, take a break
- **explain** the code to
  - someone else
  - a rubber ducky



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 ***

DON'T

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

DO

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

Exceptions

Exceptions and assertions

- what happens when procedure execution hits an **unexpected condition**?
- get an **exception**… to what was expected
  - trying to access beyond list limits

```
test = [1,7,4]
test[4]                      → IndexError
```

  - trying to convert an inappropriate type

```
int(test)                    → TypeError
```

  - referencing a non-existing variable

```
a                            → NameError
```

  - mixing data types without coercion

```
'a'/4                        → TypeError
```

OTHER TYPES OF EXCEPTIONS

- already seen common error types:
  - `SyntaxError`: Python can't parse program
  - `NameError`: local or global name not found
  - `AttributeError`: attribute reference fails
  - `TypeError`: operand doesn't have correct type
  - `ValueError`: operand type okay, but value is illegal
  - `IOError`: IO system reports malfunction (e.g. file not found)

Dealing with exceptions

- Python code can provide **handlers** for exceptions

```
try:
    a = int(input("Tell me one number:"))
    b = int(input("Tell me another number:"))
    print(a/b)
except:
    print("Bug in user input.")
```

- exceptions **raised** by any statement in body of **try** are **handled** by the **except**statement and execution continues with the body of the exceptstatement

Handling specific exceptions

- have **separate exceptclauses** to deal with a particular type of exception

```
try:
    a = int(input("Tell me one number: "))
    b = int(input("Tell me another number: "))
    print("a/b = ", a/b)
    print("a+b = ", a+b)
except ValueError:
    print("Could not convert to a number.")
except ZeroDivisionError:
    print("Can't divide by zero")
except:
    print("Something went very wrong.")
```

only execute if these errors come up

for all other errors

Other exceptions

- `else:`
  - body of this is executed when execution of associated
  - `try` body **completes with no exceptions**
- `finally`:
  - body of this is **always executed** after `try`, `else` and `except` clauses, even if they raised another error or executed a break, continueor return
  - useful for clean-up code that should be run no matter what else happened (e.g. close a file)

What to do with exceptions?

- what to do when encounter an error?
- **fail silently**:
  - substitute default values or just continue
  - bad idea! user gets no warning
- return an **“error” value**
  - what value to choose?
  - complicates code having to check for a special value
- stop execution, **signal error** condition
  - in Python: **raise an exception**

```
raise Exception("descriptive string")
```

Exceptions as control flow

- don’t return special values when an error occurred and then check whether ‘error value’ was returned
- instead, **raise an exception** when unable to produce a result consistent with function’s specification

```
raise <exceptionName>(<arguments>)
```

```
raise ValueError("something is wrong")
```

keyword

name of error you want to raise

optional, but typically a string with a message

Example: raising an exception

```
def get_ratios(L1, L2):
    """ Assumes: L1 and L2 are lists of equal length of numbers
    Returns: a list containing L1[i]/L2[i] """
    ratios = []
    for index in range(len(L1)):
        try:
            ratios.append(L1[index]/L2[index])
        except ZeroDivisionError:
            ratios.append(float('nan')) #nan = not a number  except:
            raise ValueError('get_ratios called with bad arg')
    return ratios
```

manage flow of program by raising own error

Example of exceptions

- assume we are **given a class list** for a subject: each entry is a list of two parts
  - a list of first and last name for a student
  - a list of grades on assignments

```
test_grades = [[['peter', 'parker'], [80.0, 70.0, 85.0]],
[['bruce', 'wayne'], [100.0, 80.0, 74.0]]]
```

- create a **new class list**, with name, grades, and an average

```
[[['peter', 'parker'], [80.0, 70.0, 85.0], 78.33333],
[['bruce', 'wayne'], [100.0, 80.0, 74.0], 84.666667]]]
```

Example code

```
[[['peter', 'parker'], [80.0, 70.0, 85.0]],
[['bruce', 'wayne'], [100.0, 80.0, 74.0]]]
```

```python
def get_stats(class_list):
    new_stats = []
    for elt in class_list:
        new_stats.append([elt[0], elt[1], avg(elt[1])])
    return new_stats

def avg(grades):
    return sum(grades)/len(grades)
```

Error if no grade for a student

- if one or more students **don't have any grades**, get an error

```
test_grades = [[['peter', 'parker'], [10.0, 5.0, 85.0]], [['bruce', 'wayne'], [10.0, 8.0, 74.0]],
[['captain', 'america'], [8.0,10.0,96.0]],  [['deadpool'], []]]
```

- get `ZeroDivisionError: float division by zero`

because try to

```
return sum(grades)/len(grades)
```

length is 0

Option 1: flag the error by printing a message

- decide to **notify** that something went wrong with a msg

```
def avg(grades):
    try:
        return sum(grades)/len(grades)
    except ZeroDivisionError:
        print('warning: no grades data')
```

- running on some test data gives

```
warning: no grades data
```

flagged the error

```
[[['peter', 'parker'], [10.0, 5.0, 85.0], 15.41666666],
[['bruce', 'wayne'], [10.0, 8.0, 74.0], 13.83333334],
[['captain', 'america'], [8.0, 10.0, 96.0], 17.5],
[['deadpool'], [], None]]
```

because `avg` did not return anything in the except

Option 2: change the policy

- decide that a student with no grades gets a **zero**

```
def avg(grades):
    try:
        return sum(grades)/len(grades)
    except ZeroDivisionError:
        print('warning: no grades data')
        return 0.0
```

- running on some test data gives

```
warning: no grades data
```

still flag the error

```
[[['peter', 'parker'], [10.0, 5.0, 85.0], 15.41666666],
[['bruce', 'wayne'], [10.0, 8.0, 74.0], 13.83333334],
[['captain', 'america'], [8.0, 10.0, 96.0], 17.5],
[['deadpool'], [], 0.0]]
```

now avg returns 0

Assertions

Assertions

- want to be sure that **assumptions** on state of computation are as expected
- use an **assertstatement** to raise an
- `AssertionError exception` if assumptions not met
- an example of good **defensive programming**

Example

- raises an `AssertionError` if it is given an empty list for grades
- otherwise runs ok

```
def avg(grades):
    assert len(grades) != 0, 'no grades data'
    return sum(grades)/len(grades)
```

function ends immediately if assertion not met

Assertions as defensive programming

- assertions don't allow a programmer to control response to unexpected conditions
- ensure that **execution halts** whenever an expected condition is not met
- typically used to **check inputs** to functions, but can be used anywhere
- can be used to **check outputs** of a function to avoid propagating bad values
- can make it easier to locate a source of a bug

Where to use assertions?

- goal is to spot bugs as soon as introduced and make clear where they happened
- use as a **supplement** to testing
- raise **exceptions** if users supplies **bad data input**
- use **assertions** to
  - check **types** of arguments or values
  - check that **invariants** on data structures are met
  - check **constraints** on return values
  - check for **violations** of constraints on procedure (e.g. no duplicates in a list)

References

1. MIT Introduction to Computer Science and Programming in Python

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you for your attention!