# Lecture 7: Testing, Debugging, Exceptions, Assertions

**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 own 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/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 errors for you
- have a set of expected values
  - an input set
  - for each input, the expected output.

## Classes of Tests (3 classes)

- 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):
        """ Assume x and y are ints
        Return True if y is less than x, else False """
```

natural boundary

  - Can we come up with some natural partitions?
- if no natural partitions might do random testing
  - probability that the code is correct increases with more tests.

- black box testing
  - explore paths through specifications.
- glass box testing
  - explore paths through code.

↔ given docstring coming up with test cases

## Black Box Testing

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

- designed without looking at the code
- can be done by someone other than the implementer to avoid some implementer bais
- testing can be reduced or reused if implementation changes
- path through specification
  - build test cases in different natural space partitions
  - also consider boundary conditions
    - empty lists, singleton list, large numbers, small numbers.

# Glass Box testing

- use code directly to guide design to 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 arbitrary number of times
  - missing path
- guidlines
  - branches: exercise all parts of a condition
  - for loops: loop not entered, loop entered once or more
  - while loops: same as for loop, case that catch all ways to exit loop

```
>>> def abs(x):
    """Assume x is an int
    Return 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 suit: 2 and -2
- but abs(-1) incorrectly returns -1
- should still test boundary cases

## Debugging

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

## Print Statement

- good way to test hypothesis
- when to print
  - enter function
  - parameters
  - function results
- use bisection method
  - 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 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 Examples

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

## Logic Errors - Tricky

- think before writing new code
- draw pictures, different approach for pictures
- explain the code to someone else or thy self.

### Don't

- Write entire programs
- Test entire programs
- Debug entire program
- Change code
- Remember where bug was
- Test code
- Forget where bug was or what change you made
- Panic

### Do

- Write a function
- Test function, debug function
- Repeat: write function
- Test/Debug function
- Backup then change code
- Write potential bugs in comm
- Test code
- Compare new version with old version of code

## 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, 2, 3]
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

- SyntaxError: Python can't understand 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
  - i.e. file not found

→

## Dealing with Exceptions

- Python code can provide handlers from exceptions

```
>>> try:
        a = int(input("Tell me one number: "))
        b = int(input("Tell me another number: "))
    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 except statement

## Handling Specific Exceptions

- have seperate except clauses 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("Couldnat convert to a number.")
    except ZeroDivisionError:
        print("Can't divide by zero")
    except:
        print("Something went very wrong")
```

only execute if these error come up

to catch 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, continue, or return.
    - useful for clean-up that should be run no matter what else happened (eg. closed a file)

## What to do with Exceptions?

- what to do when encounter an error?
- fail silently
  - substitute user's value with default value or just continue
    - bad idea! user gets, because user input different than what user actually inputted.
- return an "error" value
  - what value to choose?
  - complicated code having to check for a special value
- stop execution, signal error condition
  - in Python: raise an exception

raise Exception("descriptive string")

## Exception as Control Flow

- don't return special value 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

to:

raise <exceptionName>(<arguments>)

raise ValueError ("something went wrong")

keyword — Name of Error — description of error

## Example: Raising an Exception

```
>>> def get_ratio(L1, L2):
    """Assum: L1 and L2 are lists of equal length of numbers
    Return: a list containing L1[i]/L2[i]"""
    ratio = []
    for i in range(len(L1)):
        try:
            ratio.append(L1[i]/L2[i])
        except ZeroDivisionError:
            ratio.append(float('nan')) #nan = not a number
        except:
            raise ValueError('Bad argument')
    return(ratio)
```

## 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, 70, 100]],
               [['bruce', 'wayne'], [100, 80, 74]]]
```

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

```
[[['peter', 'parker'], [80, 70, 100], 83.33]],
 [['bruce', 'wayne'], [100, 80, 74], 84.67]]]
```

```
>>> def get_stats(class_list):
        new_stats = []
        for i in class_list:
            new_stats.append([i[0], i[1], avg(i[1])])
        return new_stats
>>> def avg(grades):
        return sum(grades)/len(grades)
```

## Assertions

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

## Example:

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

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

## Assertions As Defensive Programming

- assertions don't allow a programer to control response to unexpected conditions
- ensure that execution halts whenever an expected condition is not met
- tippically 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

## When to Use Assertions?

- goal is to spot bugs as soon as introduced and make clear when they happened
- use as a supplement to testing
- raise exceptions if user supplies bad data input
- use assertions: ~~if users~~
  - 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 (no duplicates in a list)