COMP10001 Foundations of Computing
Debugging and Exception Handling

Semester 2, 2016

Chris Leckie

August 28, 2016

Lecture Agenda

- Last lecture:
  - Testing and debugging
- This lecture:
  - Code diagnostics/logging
  - Using the stack for debugging
  - Assertions
  - Exceptions

Debugging

- Yesterday, we talked about classifying and avoiding errors in general terms, but actually identifying and fixing them can be a tricky/tedious business
- Debugging approaches:
  - diagnostic `print` diagnostics
  - inspect the run state of your program with `code`

Diagnostic `print` Statements

- The simplest way to better understand what your code is doing is by inserting `print` statements throughout your code, and examining the outputs:

```
def load(filename):
    """generate a list of lists from a CSV file"""
    file = open(filename)
    table = []
    for row in file.readlines():
        row = row.strip()
        fields = row.split(",")
        table.append(fields)
    return(table)
```

Diagnostic `print` Statements

- The simplest way to better understand what your code is doing is by inserting `print` statements throughout your code, and examining the outputs:

```
def load(filename):
    """Generate a list of lists from a CSV file"""
    print("In: load_csv({})".format(filename))
    file = open(filename)
    table = []
    for row in file.readlines():
        print("Processing: {}".format(row))
        row = row.strip()
        fields = row.split(",")
        table.append(fields)
        print("Table size: {}".format(len(table)))
    print ("Finished load_csv")
    return(table)
```

A Slightly Better Version

- While diagnostic `print` statements can help identify bugs, they also litter the code, and ultimately need to be taken back out
- A slightly better approach is to define a "verbosity" level, and print accordingly

A Slightly Better Version

```
def load(filename,verbosity=0):
    """generate a list of lists from a CSV file"""
    if verbosity >= 1: print("In: load_csv({})".format(filename))
    file = open(filename)
    table = []
    for row in file.readlines():
        if verbosity >= 2: print("Processing: "+row)
        row = row.strip()
        fields = row.split(",")
        table.append(fields)
        if verbosity >= 2:
            print("Table size: {}".format(len(table)))
    if verbosity >= 1: print("Finished load_csv")
    return(table)
```

A Better Version Again I

- A better version again with more functionality, is the `logging` library:

```
import logging

logging.basicConfig(level=logging.DEBUG)

def load(filename):
    """generate a list of lists from a CSV file"""
    logging.debug("In: load_csv({0})".format(filename))
    file = open(filename)
    table = []
    for row in file.readlines():
        logging.debug("Processing: {0}".format(row))
        row = row.strip()
        fields = row.split(",")
        table.append(fields)
        logging.debug("Table size: {0}".format(len(table)))
    logging.debug("Finished load_csv")
    return(table)
```

A Better Version Again II

- `logging` has the following in-built levels:
  `DEBUG` < `INFO` < `WARNING` < `ERROR` < `CRITICAL`
- `logging.basicConfig(logging.LEVEL)` defines the level above which to display log messages for
- It is also possible to suppress all messages below a certain level with:
  `logging.disable(logging.LEVEL)`
- Log messages can also be directed to a file:
  `logging.basicConfig(filename=FILE, filemode='w')`

And Now for Something Completely Different ...

- Perform each of the following tasks, as commanded by your "programmer":
  - count from 1 to 10
  - spell *computing* backwards
  - hop on your left leg 10 times
  - recite the following lines from Shakespeare:
    The quality of mercy is not strain'd,
    It droppeth as the gentle rain from heaven
    Upon the place beneath
- Perform each task on demand, interrupting the current task when asked to perform the next task, and returning to when other tasks are done

The Stack is Your Friend I

- The stack trace in the message for run-time errors can often give you valuable hints on the cause of a bug:

```
1  def tofloat(i):
2      return(flt(i))
3
4  def addnums(numlist):
5      total = 0
6      for i in numlist:
7          total += tofloat(i)
8      return(total)
9
10 nums = [1,2,3]
11 addnums(nums)
```

The Stack is Your Friend II

```
Traceback (most recent call last):
  File "<web session>", line 1, in <module>
  File "buggy-basic.py", line 11, in <module>
    addnums(nums)
  File "buggy-basic.py", line 7, in addnums
    total += tofloat(i)
  File "buggy-basic.py", line 2, in tofloat
    return flt(i)
NameError: global name 'flt' is not defined
```

From this, we can reproduce the sequence in which the functions were called, and *how* they were called, to be able to isolate the problem

Inspecting the Run State with code

- Just as it is often useful to be able to access variables from the console after code execution, we can suspend the execution of code and examine the variable values with `code`:

```
import code

def tofloat(i):
    code.interact(local=dict(globals(), **locals()))
    return(flt(i))

def addnums(numlist):
    total = 0
    for i in numlist:
        total += tofloat(i)
    return(total)

nums = [1,2,3]
addnums(nums)
```

Aside: starred arguments

```
def add(a=0, b=0):
    return(a+b)

lst = [1,2]
dic = d = { "b": 16 , "a": 4 }

print(add())      # 0
print(add(lst))   # error
print(add(*lst)) # 3
print(add(**d))   # 20
```

- * in front of an iterable argument asks for the iterable to be "unpacked" into individual arguments
- ** in front of a map argument asks for the iterable to be "unpacked" into individual keyword arguments

Assertions I

- To date, we have tended to assume well-behaved inputs to our functions etc., and lived with the fact that ill-behaved inputs will cause a logic or run-time error, e.g.:

```
def withdraw(amount,balance):
    if balance < -100:
        print("Insufficient balance")
        return(balance)
    else:
        print("Withdrawn")
        return(balance - amount)
>>> withdraw(100,False)
Withdrawn
-100
```

Assertions II

- One way to ensure that the inputs are of the right type is with `assert`:

```
def withdraw(amount,balance):
    assert type(balance) == int
    if balance < -100:
        print("Insufficient balance")
        return(balance)
    else:
        print("Withdrawn")
        return(balance - amount)
>>> withdraw(100,'a')
Traceback (most recent call last):
...
AssertionError
```

Assertions III

- Note, however, that assertions should be used sparingly and reserved for "impossible" code states
- Use an explicit `if` statement if the result is important to the logic of the code

Exception Handling I

- We are used to seeing python "raise exceptions" as a result of run-time errors:

```
>>> 9/0
Traceback (most recent call last):
  File "<web session>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero

>>> 1 + "2"
Traceback (most recent call last):
  File "<web session>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'

>>> 1 + i
Traceback (most recent call last):
  File "<web session>", line 1, in <module>
NameError: name 'i' is not defined
```

Exception Handling II

- Other common run-time exceptions are:
  - `AssertionError`: raised when an `assert` fails
  - `IndexError`: raised when an index is out of range
  - `KeyError`: raised when a key is not found in a dictionary
- It is possible to "handle" exceptions within your code using `try ... except ...`.
- `try` attempts to execute its block of code, and passes off to the exception handlers (which are also tested in linear order) only if an exception is raised during the execution, before running the code block attached to `finally`

Exception Handling III

For example:

```
while True:
    try:
        x = int(input("Please enter a number: "))
        break
    except ValueError:
        print("Oops! Try again...")
```

Lecture Summary

- What is diagnostic `print` statements, and what simple improvements are there over them?
- How can the stack be used in debugging?
- What are assertions, and how/why are they used?
- What are exceptions, and how can we handle them?