6.00.1.x - Week 4 - Good Programming Practices

October 11, 2020

1 Video 1: Programming Challenges

Three strategies to keep your code free of bugs:

1. Testing.
    - Compare input/output pairs to specification.
    - How can I break my program?
2. Defensive programming.
    - Write specifications for functions.
    - Modularize programs.
    - Check conditions on inputs/outputs (assertions).
3. Debugging.
    - Study events leading up to an error.
    - "Why is it not working?"
    - "How can I fix my program?"

2 Video 2: Classes of Tests

Setting yourself up for easy testing and debugging:

- From the **start**, design ode to ease this part.
- Break program into **modules** that can be tested and debugged individually.
- Docuemnt constraints on modules.
- Document assumptions behind code design.

When are you ready to test?

- Ensure **code runs**.
- Have a **set of expected results**.

Classes of tests

- Unit testing.
  - Validate each piece of program.
  - Testing each function separately.
- Regression testing.
  - Add test for bugs as you find them in a function.
  - Catch reintroduced errors that were previously fixed.
- Integration testing.
  - Does overall program work?
  - DON'T rush to do this, but do it eventually.

Testing approaches

- **Intuition** about natural boundaries to the problem.
  - Can you come up with some natural partitions?
- If no natural partitions, might do **random testing**.
- **Black box testing**.
  - 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 especification.
- **Glass box testing**.

- **Use code** directly to guide design of test cases.
- Called **path complete** if every potential through code is tested at least once.
- Drawbacks: it can go through loops arbitrarily many times; you may also miss paths.
- A path complete test suite could **miss a bug**.

In glass box testing, we try to sample as many paths through the code as we can. In the case loops, we want to sample three cases:

1. Not executing the loop at all.
2. Executing the loop exactly once.
3. Executing the loop multiple times.

3 Video 3: Bugs

Once you know there are bugs, you will want to:

- Isolate the bugs.
- Eradicate the bugs.
- Retest until code runs correctly.

Overt vs. covert

- Overt has an obvious manifestation - code crashes or runs forever.
- Covert has no obvious manifestation - code returns a value, which may be incorrect but hard to determine.

Persistent vs. intermittent

- Persistent occurs every time code is run.
- Intermittent only occurs some times, even if run on same input.

4 Video 4: Debugging

Some tools for debugging:

- Built in to IDLE and Anaconda.
- Python Tutor.
- print statement.
- use your brain, be systematic in your hunt.

Print statements

- When to print:
  - enter function.
  - parameters.
  - function results.
- Use bisection method:
  - put print halfway in code.
  - decide where bug may be depending on values.

Types of errors:

- IndexError
- TypeError
- NameError
- SyntaxError

Logic errors are hard:

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

Debugging steps

- Study program code.
- Use the **scientific method** in order to debug.

5 Video 5: Debugging Example

Treat as a search problem: looking for explanation for incorrect behavior.

- Study available data.
- Form an hypothesis consistent with the data.
- Design and run a repeatable experiment with potential to refute the hypothesis.
- Keep record of experiments performed: use narrow range of hypotheses.

6 Video 6: Exceptions

Exceptions

What happens when procedure execution hits an **unexpected condition**?

Get an exception to what was expected:

- Trying to access beyond list limits.
- Trying to convert an inappropriate type.
- Referencing a non-existing variable.
- Mixing data types without coercion.

Other types of exceptions:

- SyntaxError.
- NameError.
- AttributeError.
- TypeError.
- ValueError.
- IOError.

What to do with exceptions?

- Fail silently: bad idea. User gets no warning.
- Return an **error** value: complicates code having to check for a special value.
- Stop execution, **signal error** condition. In Python: **raise an exception**.

[2]:
```
raise Exception("descriptive String")
```

```
       ␣
↪---------------------------------------------------------------------------

       Exception                                 Traceback (most recent call␣
↪last)

       <ipython-input-2-6295980b03e8> in <module>
   ----> 1 raise Exception("descriptive String")


       Exception: descriptive String
```

```
[3]: try:
         a = int(input('Tell me one number:'))
         b = int(input('Tell me another number:'))
         print(a/b)
         print ('Okay')
     except:
         print('Bug in user input.')
     print('Outside')
```

```
Tell me one number:0
Tell me another number:1
0.0
Okay
Outside
```

Exceptions **raised** by any statement in body of **try** are **handled** by the **except** statement and execution continues after the body of the except statement. Very useful!

This can also be done saying the type of error after the except, to distinguish between the different exceptions that might come up.

Other options are:

- else: body of this is executed when executions 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 rror or executed a break, continue or return. This is useful for clean-up code that should be run no matter what else happened (e.g. close a file).

7 Video 7: Exceptions Examples

First example of how to use the structure mentioned in the video 6.

[4]:
```
while True:
    try:
        n = input('Please enter an integer: ')
        n = int(n)
        break
    except ValueError:
        print('Input not an integer; try again')
print('Correct input of an integer!')
```

```
Please enter an integer: 2
Correct input of an integer!
```

This only handles ValueErrors.

A really nice example of how to use this to actually implement code is shown in the following case.

[5]:
```
data = []
file_name = input('Provide a name of a file of data ')

try:
    fh = open(file_name, 'r')
except IOError:
    print('cannot open', file_name)
else:
    for new in fh:
        if new != '\n':
            addIt = new[:-1].split(',') #remove trailing \n
            data.append(addIt)
finally:
    fh.close() # close file even if fail

gradesData = []
if data:
    for student in data:
        try:
            name = student[0:-1]
            grades = int(student[-1])
            gradesData.append([name, [grades]])
        except ValueError:
            gradesData.append([student[:], []])
```

```
Provide a name of a file of data a
cannot open a
```

```
      ␣
↪---------------------------------------------------------------------------

       NameError                                 Traceback (most recent call␣
↪last)

       <ipython-input-5-ac0d75feac4c> in <module>
        12             data.append(addIt)
        13 finally:
   ---> 14     fh.close() # close file even if fail
        15
        16 gradesData = []


       NameError: name 'fh' is not defined
```

8 Video 8: Exceptions as Control Flow

A new keyword: raise.

- 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.

Example:

```
[6]: def get_ratios(L1, L2):
         ratios = []
         for index in range(len(L1)):
             try:
                 ratios.append(L1[index]/float(L2[index]))
             except ZeroDivisionError:
                 ratios.append(float('NaN')) #NaN = Not a Number
             except:
                 raise ValueError('get_ratios called with bad arg')
         return ratios

     L1 = [1,2,3,4]
     L2 = [5,6,7,8]
     get_ratios(L1,L2)
```

```
[6]: [0.2, 0.3333333333333333, 0.42857142857142855, 0.5]
```

```
[7]: L3 = [5,6,7]
     get_ratios(L1,L3)
```

```
            ␣
     ↪---------------------------------------------------------------------------

        IndexError                                Traceback (most recent call␣
     ↪last)

        <ipython-input-6-049f68bfb879> in get_ratios(L1, L2)
          4         try:
    ----> 5             ratios.append(L1[index]/float(L2[index]))
          6         except ZeroDivisionError:


        IndexError: list index out of range


    During handling of the above exception, another exception occurred:
```

```
    ValueError                                Traceback (most recent call
↪last)

    <ipython-input-7-f74906737b77> in <module>
      1 L3 = [5,6,7]
----> 2 get_ratios(L1,L3)


    <ipython-input-6-049f68bfb879> in get_ratios(L1, L2)
      7             ratios.append(float('NaN')) #NaN = Not a Number
      8         except:
----> 9             raise ValueError('get_ratios called with bad arg')
     10     return ratios
     11


    ValueError: get_ratios called with bad arg
```

```
[8]: L4 = [5,6,7,0]
     get_ratios(L1,L4)
```

```
[8]: [0.2, 0.3333333333333333, 0.42857142857142855, nan]
```

9 Video 9: 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**.

Really interesting idea! Let's see an example.

```
[10]: def avg(grades):
          assert not len(grades) == 0, 'no grades data'
          return sum(grades)/len(grades)
```

```
[11]: avg([1,2,3])
```

```
[11]: 2.0
```

```
[12]: avg([])
```

```
       ␣
 ↪---------------------------------------------------------------------------

       AssertionError                            Traceback (most recent call␣
 ↪last)

       <ipython-input-12-938d6a1744f6> in <module>
   ----> 1 avg([])


       <ipython-input-10-2221a6b79207> in avg(grades)
         1 def avg(grades):
   ----> 2     assert not len(grades) == 0, 'no grades data'
         3     return sum(grades)/len(grades)


       AssertionError: no grades data
```

- Raises an AssertionError if it is given an empty list for grades, otherwise runs ok.

Properties of assertions

- 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 fuctions procedures, 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?

- Check types of arguments or values (e.g. no string stresses).
- Check that invariants on data structures are met.
- Check constraints on return values (e.g. no negative distances).
- Check for violations of constraints on procedure (e.g. no duplicates in a list).

10 Notes during Problem Set #4

A more organized way of checking whether a key is or not in a dictionary is shown.

```
[ ]: hand = {'a':1, 'q':1, 'l':2, 'm':1, 'u':1, 'i':1}
     hand['e']
```

```
[ ]: hand.get('e', 0)
```