Documentation, testing and debugging

- docstring
- defensive programming
- assert
- test driven developement
- assertions
- testing
- unittest
- debugger

What is good code ?

- Readability
  - well-structured
  - documentation
  - comments
  - follow some standard structure (easy to recognize, follow PEP8 Style Guide)

- Correctness
  - outputs the correct answer on valid input
  - eventually stops with an answer on valid input (should not go in infinite loop)

- Reusable...

Why ?

Documentation

- *specification of functionality*
- docstring
  - *for users of the code*
  - modules
  - methods
  - classes
- comments
  - *for readers of the code*

Testing

- Correct implementation ?
- Try to predict behavior on unknown input ?
- Performance guarantees ?

Debugging

- *Where is the #!¤$ bug ?*

”Program testing can be used to show the presence of bugs, but never to show their absence” --- Edsger Dijkstra

Built-in exceptions (class hierarchy)

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      |
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      |    +-- RecursionError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarning
```

docs.python.org/3/library/exceptions.html

Testing for unexpected behaviour ?

infinite-recursion1.py

```
def f(depth):
    f(depth + 1)  # infinite recursion ⚠

f(0)
```

Python shell

```
| RecursionError: maximum recursion depth exceeded
```

infinite-recursion2.py

```
def f(depth):
    if depth > 100:
        print("runaway recursion???")
        raise SystemExit  # raise built-in exception
    f(depth + 1)

f(0)
```

Python shell

```
| runaway recursion???
```

infinite-recursion3.py

```
import sys

def f(depth):
    if depth > 100:
        print("runaway recursion???")
        sys.exit()  # system function
    f(depth + 1)

f(0)
```

raises SystemExit

Python shell

```
| runaway recursion???
```

- let the program eventually fail
- check and raise exceptions
- check and call `sys.exit`

Catching unexpected behaviour – `assert`

infinite-recursion4.py

```
def f(depth):
    assert depth <= 100   # raise exception if False
    f(depth + 1)

f(0)
```

Python shell

```
| File "...\infinite-recursion4.py", line 2, in f
|     assert depth <= 100
| AssertionError
```

infinite-recursion5.py

```
def f(depth):
    assert depth <= 100, "runaway recursion???"
    f(depth + 1)

f(0)
```

Python shell

```
| File ".../infinite-recursion5.py", line 2, in f
|     assert depth <= 100, "runaway recursion???"
| AssertionError: runaway recursion???
```

- keyword `assert` checks if boolean expression is true, if not, raises exception AssertionError
- optional second parameter passed to the constructor of the exception

Disabling `assert` statements

Command Prompt

```
C:\Users\au121\Desktop>python -O infinite-recursion5.py
Traceback (most recent call last):
  File "infinite-recursion5.py", line 5, in <module>
    f(0)
  File "infinite-recursion5.py", line 3, in f
    f(depth + 1)
  File "infinite-recursion5.py", line 3, in f
    f(depth + 1)
  File "infinite-recursion5.py", line 3, in f
    f(depth + 1)
  [Previous line repeated 995 more times]
RecursionError: maximum recursion depth exceeded

C:\Users\au121\Desktop>
```

- **`assert`** statements are good to help check correctness of program – but can slow down program
- invoking Python with option `-O` disables all assertions (by setting `__debug__` to `False`)

docs.python.org/3/reference/simple_stmts.html#assert

Example

$$\left\lfloor \sqrt{x} \right\rfloor$$

First try... (seriously, the bugs were not on purpose)

intsqrt_buggy.py

```python
def int_sqrt(x):
    low = 0
    high = x
    while low < high - 1:
        mid = (low + high) / 2
        if mid ** 2 <= x:
            low = mid
        else:
            high = mid
    return low
```

Python shell

```
> int_sqrt(10)
| 3.125  # 3.125 ** 2 = 9.765625
> int_sqrt(-10)
| 0  # what should the answer be ?
```

Let us add a specification...

intsqrt.py

```
def int_sqrt(x):
    """Compute the integer square root of an integer x.

    Assumes that x is an integer and x >= 0
    Returns integer floor(sqrt(x))"""

    ...
```

docstring; input requirements; output guarantees

Python shell

```
> help(int_sqrt)
| Help on function int_sqrt in module __main__:
|
| int_sqrt(x)
|     Compute the integer square root of an integer x.
|     Assumes that x is an integer and x >= 0
|     Returns integer floor(sqrt(x))
```

- all methods, classes, and modules can have a **`docstring`** (ideally have) as a **specification**
- for methods: summarize purpose in first line, followed by input requirements and ouput guarantees
- the docstring is assigned to the object's `__doc__` attribute

PEP 257 -- Docstring Conventions
www.python.org/dev/peps/pep-0257/

Let us check input requirements...

intsqrt.py

```
def int_sqrt(x):
    """Compute the integer square root of an integer x.

    Assumes that x is an integer and x >= 0
    Returns integer floor(sqrt(x))"""

    assert isinstance(x, int)   } check input
    assert 0 <= x               } requirements
    ...
```

Python shell

```
> int_sqrt(-10)
|  File "...\int_sqrt.py", line 7, in int_sqrt
|     assert 0 <= x
| AssertionError
```

- doing explicit checks for valid input arguments is part of **defensive programming** and helps spotting errors early

  (instead of continuing using likely wrong values... resulting in a final meaningless error)

Let us check if output correct...

intsqrt.py

```
def int_sqrt(x):
    """Compute the integer square root of an integer x.

    Assumes that x is an integer and x >= 0
    Returns integer floor(sqrt(x))"""

    assert isinstance(x, int)
    assert 0 <= x
    ...
    assert isinstance(result, int)                    } check
    assert result ** 2 <= x < (result + 1) ** 2       } output
    return result
```

Python shell

```
> int_sqrt(10)
|   File "...\int_sqrt.py", line 20, in int_sqrt
|     assert isinstance(result, int)
| AssertionError
```

- output check identifies the error

  `mid = (low+high) / 2`

- should have been

  `mid = (low+high) // 2`

- The output check helps us to ensure that functions specification is guaranteed in applications

Let us test some input values...

intsqrt.py

```
def int_sqrt(x):
    ...

assert int_sqrt(0) == 0
assert int_sqrt(1) == 1
assert int_sqrt(2) == 1
assert int_sqrt(3) == 1
assert int_sqrt(4) == 2
assert int_sqrt(5) == 2
assert int_sqrt(200) == 14
```

Python shell

```
| Traceback (most recent call last):
|   File "...\int_sqrt.py", line 28, in <module>
|     assert int_sqrt(1) == 1
|   File "...\int_sqrt.py", line 21, in int_sqrt
|     assert result ** 2 <= x < (result + 1) ** 2
| AssertionError
```

- test identifies
  wrong output for x = 1

Let us check progress of algorithm...

intsqrt.py

```
...
low, high = 0, x
while low < high - 1:
    assert low ** 2 <= x < high ** 2
    mid = (low + high) / 2
    if mid ** 2 <= x:
        low = mid
    else:
        high = mid
result = low
...
```

check invariant for loop

Python shell

```
| Traceback (most recent call last):
|   File "...\int_sqrt.py", line 28, in <module>
|     assert int_sqrt(1) == 1
|   File "...\int_sqrt.py", line 21, in int_sqrt
|     assert result ** 2 <= x < (result + 1) ** 2
| AssertionError
```

- test identifies wrong output for x = 1
- but invariant apparently correct ???
- problem

  `low == result == 0`
  `high == 1`

  implies loop never entered
- output check identifies the error

  `high = x`
- should have been

  `high = x + 1`

Final program

We have used assertions to:

- Test if input arguments / usage is valid (defensive programming)
- Test if computed result is correct
- Test if an internal invariant in the computation is satisfied
- Perform a final test for a set of test cases (should be run whenever we change anything in the implementation)

intsqrt.py

```
def int_sqrt(x):
    """Compute the integer square root of an integer x.

    Assumes that x is an integer and x >= 0
    Returns the integer floor(sqrt(x))"""

    assert isinstance(x, int)
    assert 0 <= x

    low, high = 0, x + 1
    while low < high - 1:
        assert low ** 2 <= x < high ** 2
        mid = (low + high) // 2
        if mid ** 2 <= x:
            low = mid
        else:
            high = mid
    result = low

    assert isinstance(result, int)
    assert result ** 2 <= x < (result + 1) ** 2

    return result

assert int_sqrt(0) == 0
assert int_sqrt(1) == 1
assert int_sqrt(2) == 1
assert int_sqrt(3) == 1
assert int_sqrt(4) == 2
assert int_sqrt(5) == 2
assert int_sqrt(200) == 14
```

Which checks would you add to the below code?

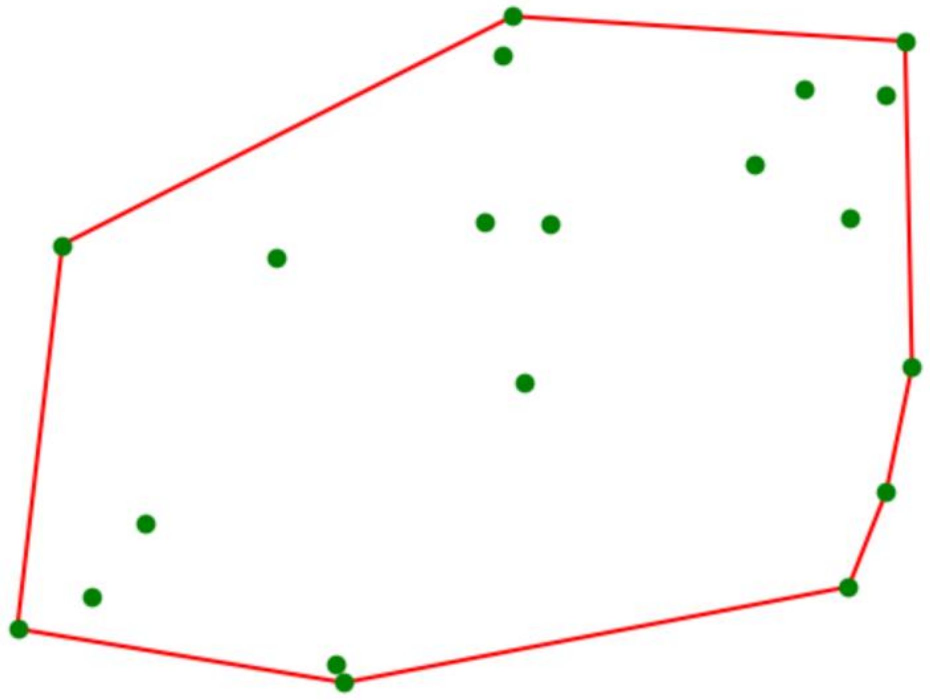
binary-search.py

```python
def binary_search(x, L):
    """Binary search for x in sorted list L

    Assumes x is an integer, and L a non-decreasing list of integers

    Returns index i, -1 <= i < len(L), where L[i] <= x < L[i+1],
    assuming L[-1] = -infty and L[len(L)] = +infty"""
    low, high = -1, len(L)
    while low + 1 < high:
        mid = (low + high) // 2
        if x < L[mid]:
            high = mid
        else:
            low = mid
    result = low
    return result
```

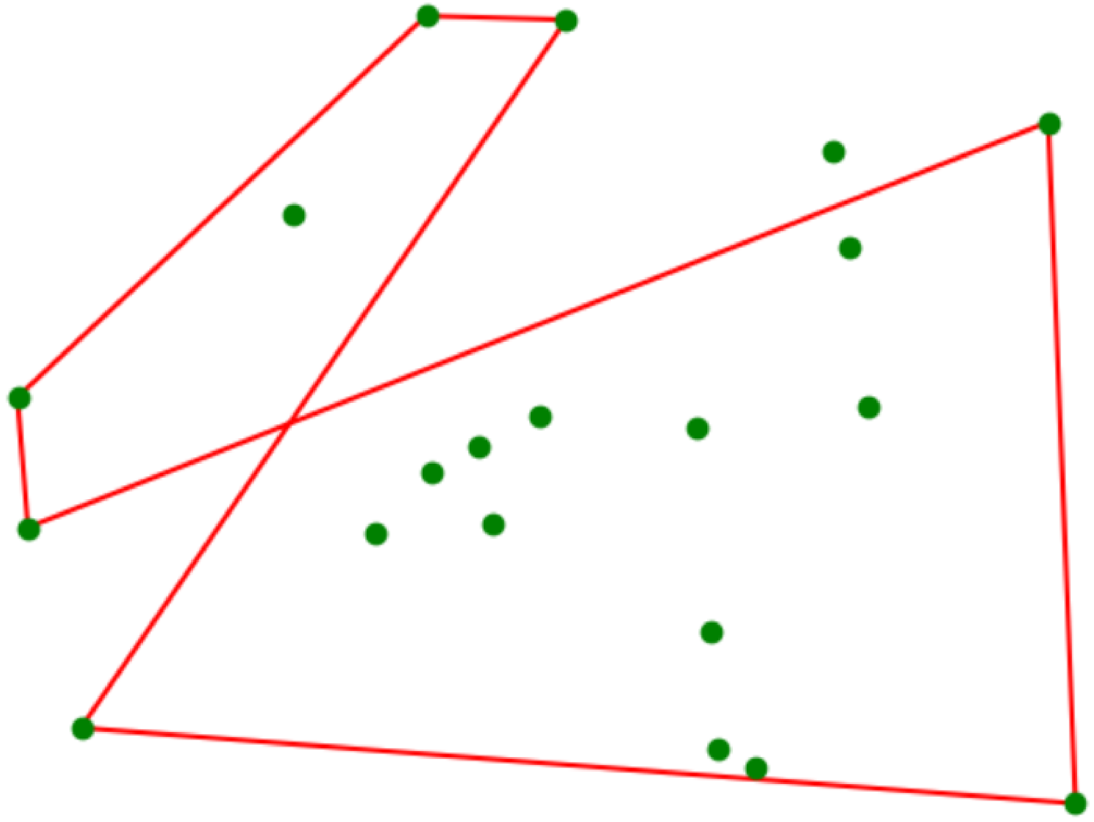
Testing – how ?

- Run set of test cases
  - test all cases in input/output specification **(black box testing)**
  - test all special cases **(black box testing)**
  - set of tests should force all lines of code to be tested **(glass box testing)**
- Visual test
- Automatic testing
  - Systematically / randomly generate input instances
  - Create function to **validate** if output is correct (hopefully easier than finding the solution)
- Formal verification
  - Use computer programs to do formal proofs of correctness, like using Coq

Visual testing – Convex hull computation



Correct



Bug !
(not convex)

doctest

- Python module
- Test instances (pairs of input and corresponding output) are written in the doc strings, formatted as in an interactive Python session

binary-search-doctest.py

```
def binary_search(x, L):
    """Binary search for x in sorted list L

    Examples:
    >>> binary_search(42, [])
    -1
    >>> binary_search(42, [7])
    0
    >>> binary_search(42, [7,7,7,56,81])
    2
    >>> binary_search(8, [1,3,5,7,9])
    3
    """

    low, high = -1, len(L)
    while low + 1 < high:
        mid = (low + high) // 2
        if x < L[mid]:
            high = mid
        else:
            low = mid
    return low

import doctest
doctest.testmod(verbose=True)
```

Python shell

```
| Trying:
|     binary_search(42, [])
| Expecting:
|     -1
| ok
| Trying:
|     binary_search(42, [7])
| Expecting:
|     0
| ok
| Trying:
|     binary_search(42, [7,7,7,56,81])
| Expecting:
|     2
| ok
| Trying:
|     binary_search(8, [1,3,5,7,9])
| Expecting:
|     3
| ok
| 1 items had no tests:
|     __main__
| 1 items passed all tests:
|    4 tests in __main__.binary_search
| 4 tests in 2 items.
| 4 passed and 0 failed.
| Test passed.
```

docs.python.org/3/library/doctest.html

unittest

- Python module
- A comprehensive object-oriented test framework, inspired by the corresponding JUnit test framework for Java

binary-search-unittest.py

```
def binary_search(x, L):
    """Binary search for x in sorted list L"""

    low, high = -1, len(L)
    while low + 1 < high:
        mid = (low + high) // 2
        if x < L[mid]:
            high = mid
        else:
            low = mid
    return low

import unittest

class TestBinarySearch(unittest.TestCase):
    def test_search(self):
        self.assertEqual(binary_search(42, []), -1)
        self.assertEqual(binary_search(42, [7]), 0)
        self.assertEqual(binary_search(42, [7,7,7,56,81]), 2)
        self.assertEqual(binary_search(8, [1,3,5,7,9]), 3)

    def test_types(self):
        self.assertRaises(TypeError, binary_search, 5, ['a', 'b', 'c'])

unittest.main(verbosity=2)
```

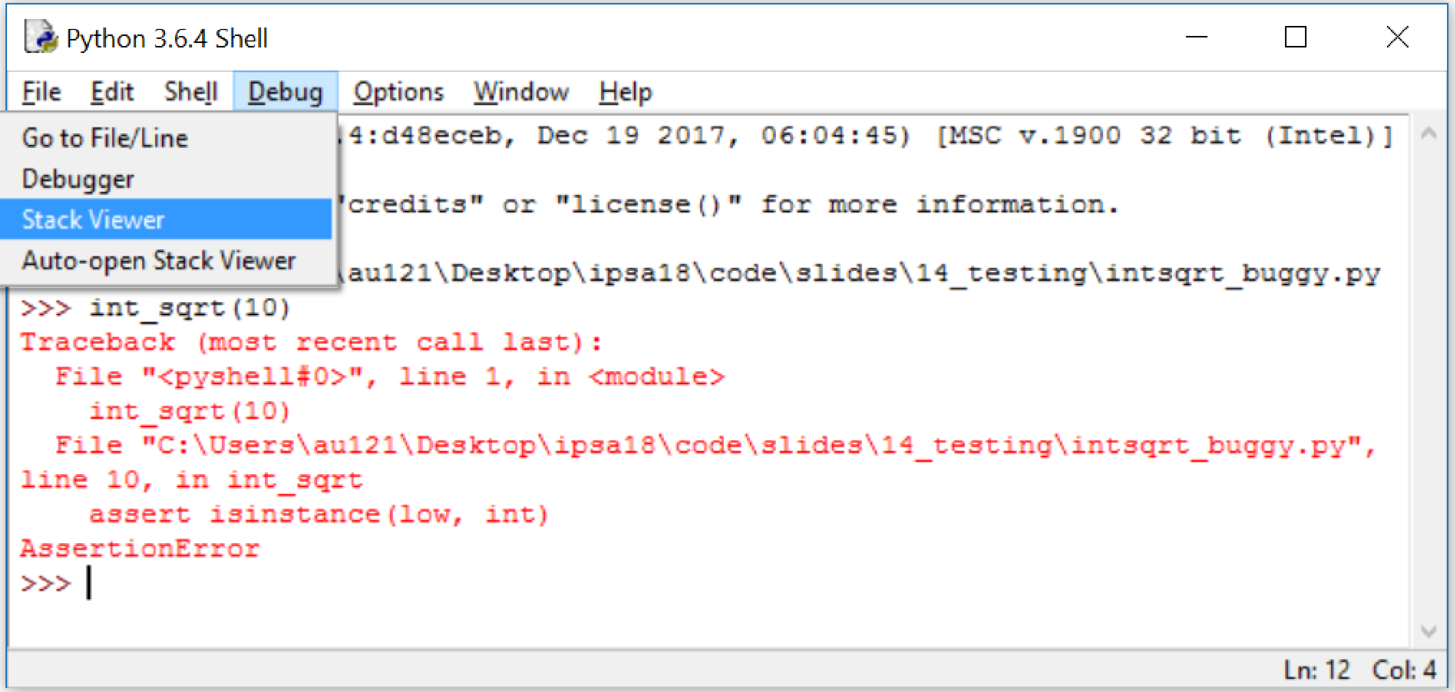
Python shell

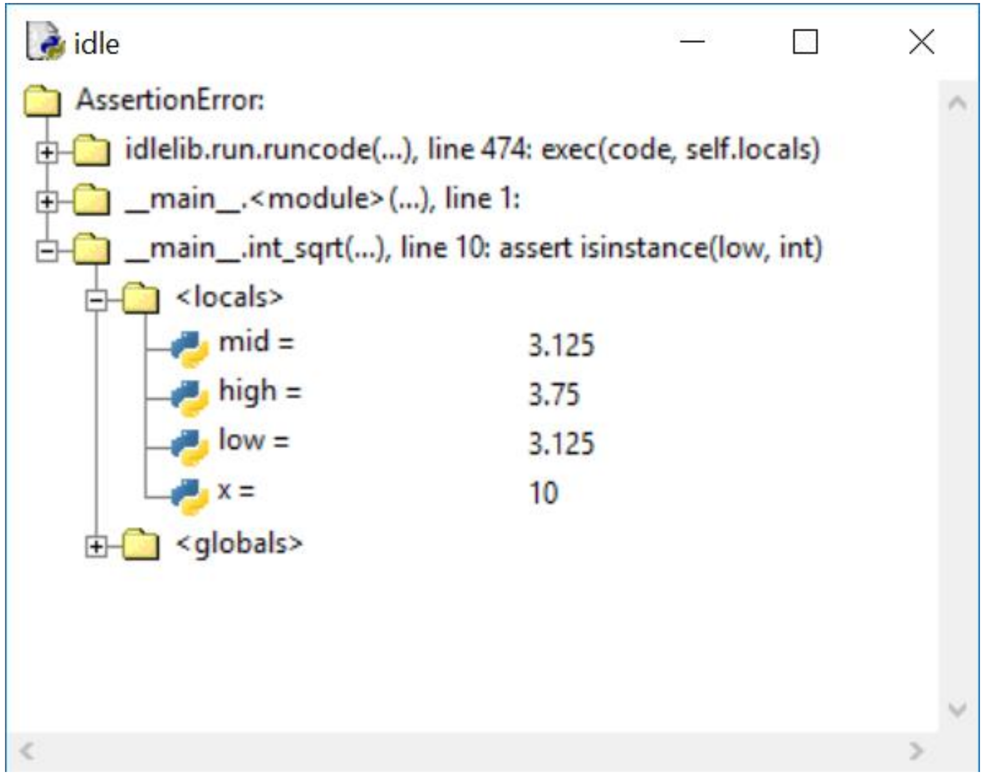
```
| test_search (__main__.TestBinarySearch) ... ok
| test_types (__main__.TestBinarySearch) ... ok
| ----------------------------------------------------------------------
| Ran 2 tests in 0.051s
| OK
```

docs.python.org/3/library/unittest.html

Debugger (IDLE)

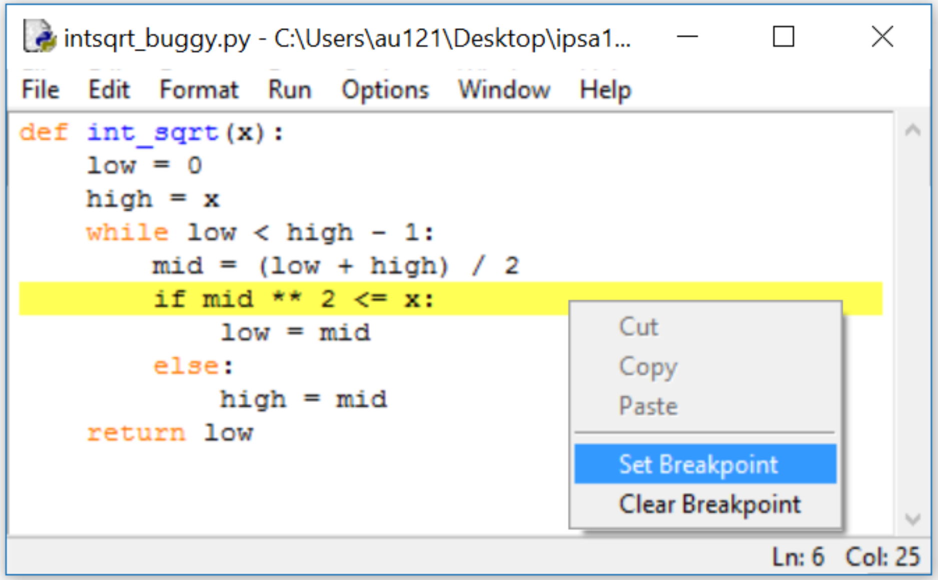
- When an exception has stopped the program, you can examine the state of the variables using **Debug > Stack Viewer** in the Python shell

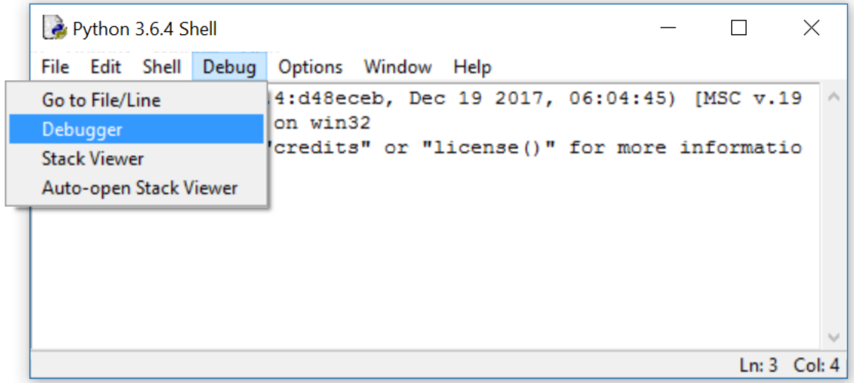


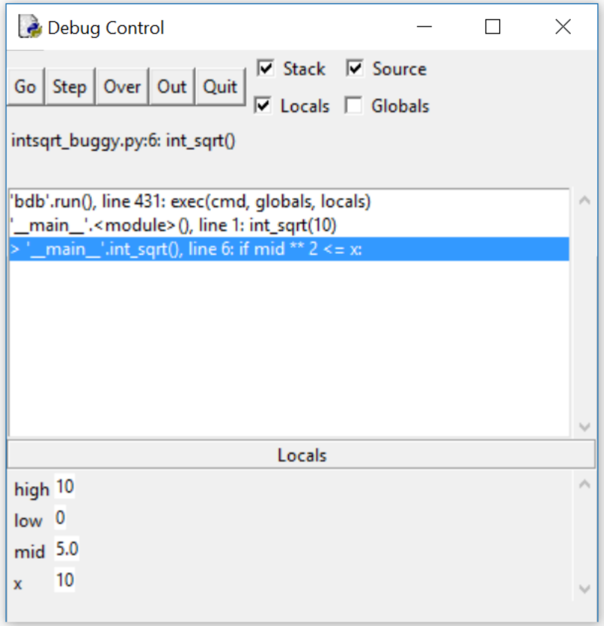


Stepping through a program (IDLE debugger)

- **Debug > Debugger** in the Python shell opens Debug Control window
- **Right click** on a code line in editor to set a "breakpoint" in your code
- **Debug Control:** Go → run until next breakpoint is encountered;
  Step → execute one line of code; Over → run function call without details;
  Out → finish current function call; Quit → Stop program;







Concluding remarks

- Simple debugging: add print statements
- **Test driven development** → Strategy for code development, where tests are written before the code
- **Defensive programming** → add tests (assertions) to check if input/arguments are valid according to specification
- When designing tests, ensure **coverage**
(the set of test cases should make sure all code lines get executed)
- **Python testing frameworks: doctest, unittest, pytest, ...**