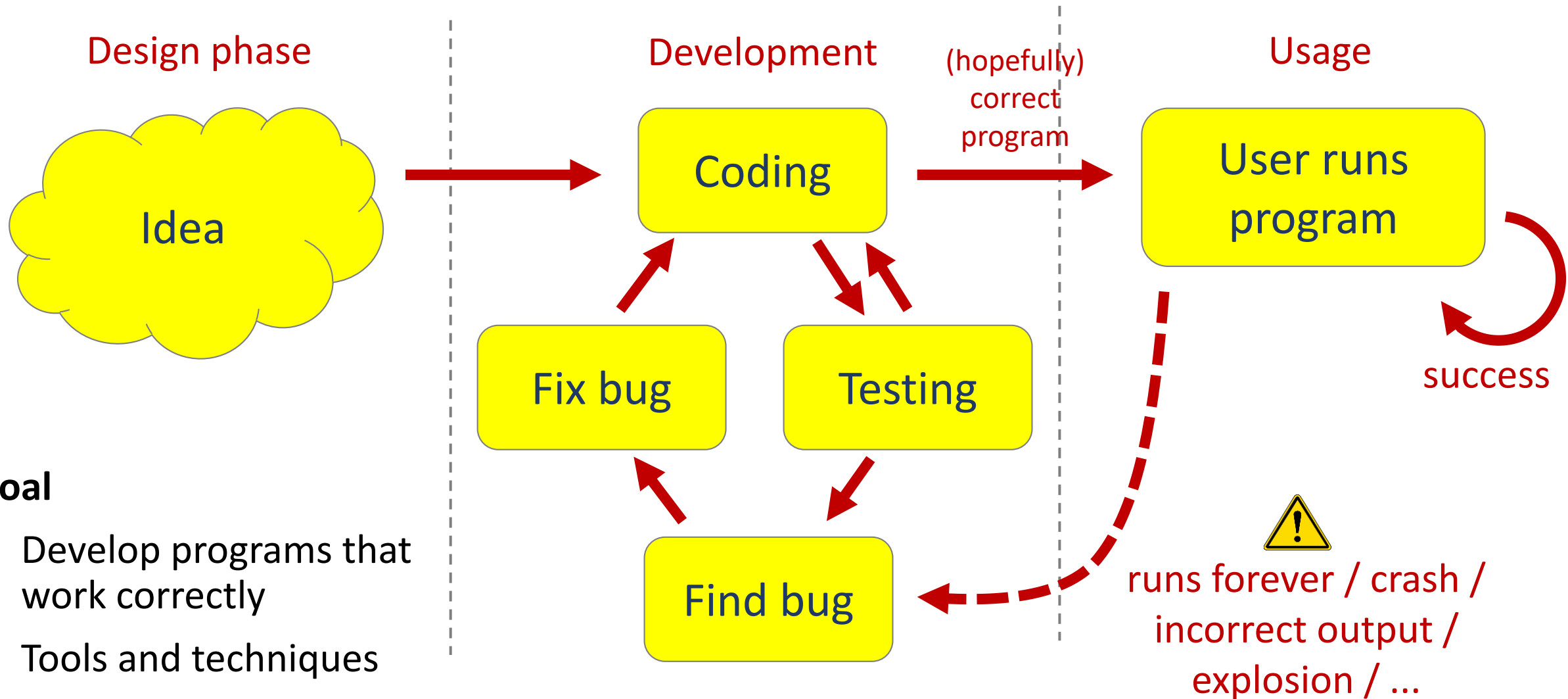


# Documentation, testing and debugging

- docstring
- defensive programming
- assert
- test driven developement
- assertions
- testing
- unittest
- debugger
- static type checking (mypy)

# Ensuring good quality code ?



## Goal

- Develop programs that work correctly
- Tools and techniques

# 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 #!x\$ bug ?*

# 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
    +-- FileNotFoundError
    +-- 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
```

# 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
```

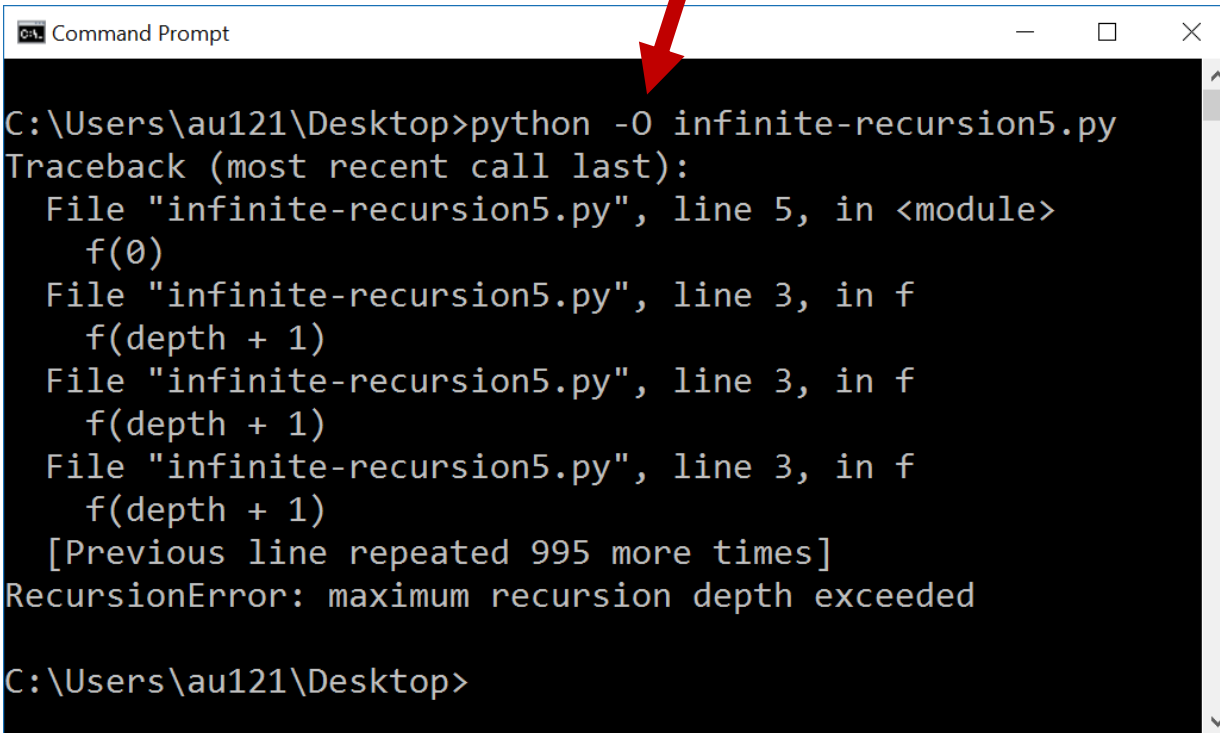
```
def f(depth):  
    assert depth <= 100, "runaway recursion???"  
    f(depth + 1)  
  
f(0)
```

```
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

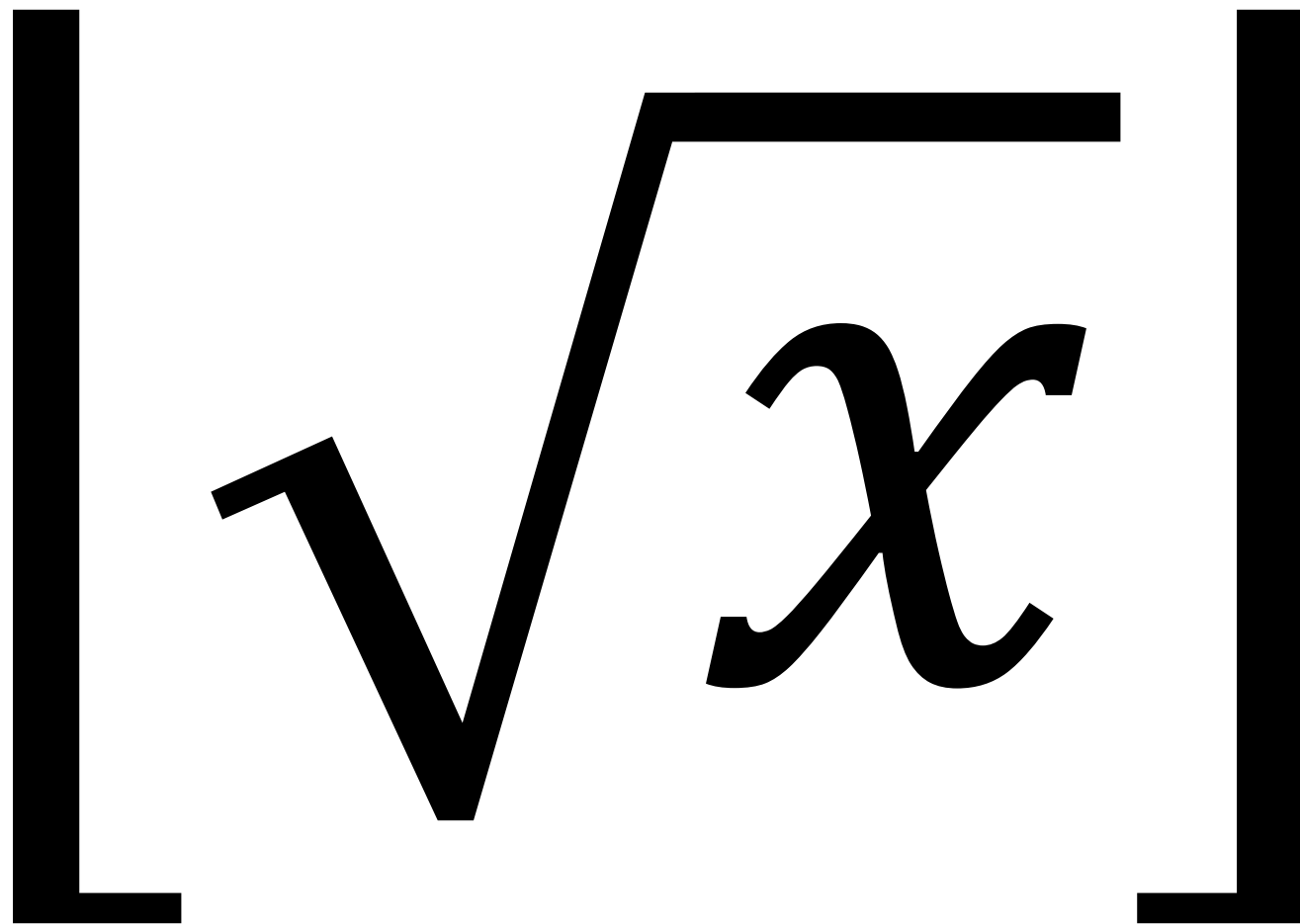


```
Command Prompt
C:\Users\au121\Desktop>python -O infinite-recursion5.py
Traceback (most recent call last):
  File "infinite-recursion5.py", line 5, in <module>
    f(0)
  File "infinite-recursion5.py", line 3, in f
    f(depth + 1)
  File "infinite-recursion5.py", line 3, in f
    f(depth + 1)
  File "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`)



Example



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

```
intsqrt_buggy.py
```

```
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 output guarantees
- the docstring is assigned to the object's `__doc__` attribute

# 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)  
    assert result ** 2 <= x < (result + 1) ** 2 } check  
    return result } output
```

`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 } check invariant
    mid = (low + high) / 2           for loop
    if mid ** 2 <= x:
        low = mid
    else:
        high = mid
result = low
...
```

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`

```
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
```

## binary-search-assertions.py

```
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"""

    input {
        assert isinstance(x, int)
        assert isinstance(L, list)
        assert all([isinstance(e, int) for e in L])
        assert all([L[i] <= L[i + 1] for i in range(len(L) - 1)])
    } inefficient 

    low, high = -1, len(L)
    loop {
        while low + 1 < high:
            assert (low == -1 or L[low] <= x) and (high == len(L) or x < L[high])
            mid = (low + high) // 2
            if x < L[mid]:
                high = mid
            else:
                low = mid
        result = low
    }

    output {
        assert ((result == -1 and (len(L) == 0 or x < L[0]))
                or (result == len(L) - 1 and x >= L[-1])
                or (0 <= result < len(L) - 1 and L[result] <= x < L[result + 1]))
    }

    return result

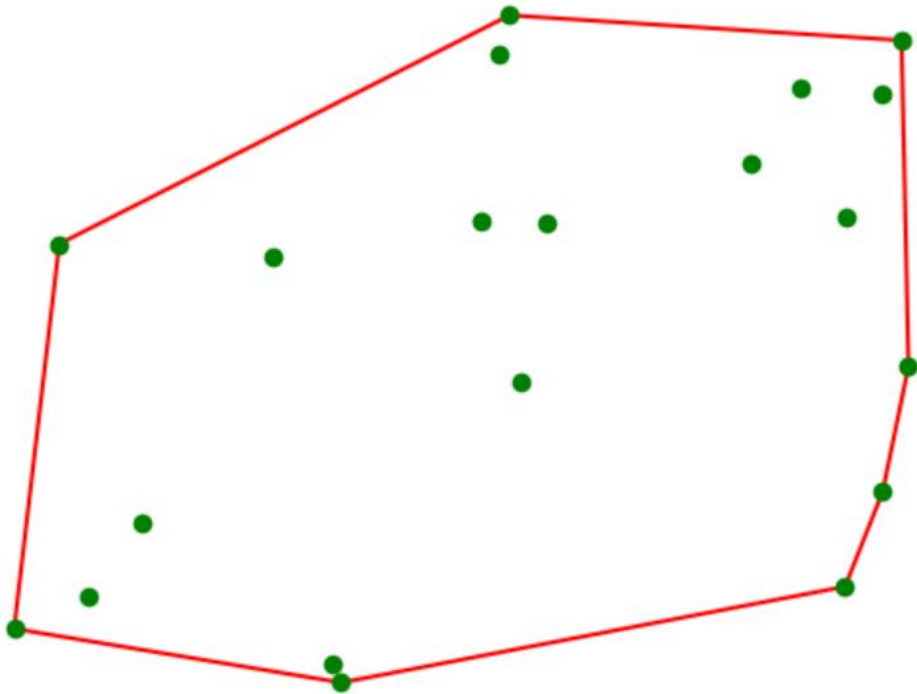
assert binary_search(42, []) == -1
assert binary_search(42, [7]) == 0
assert binary_search(7, [42]) == -1
assert binary_search(7, [42,42,42]) == -1
assert binary_search(42, [7,7,7]) == 2
assert binary_search(42, [7,7,7,56,81]) == 2
assert binary_search(8, [1,3,5,7,9]) == 3

} test cases
```

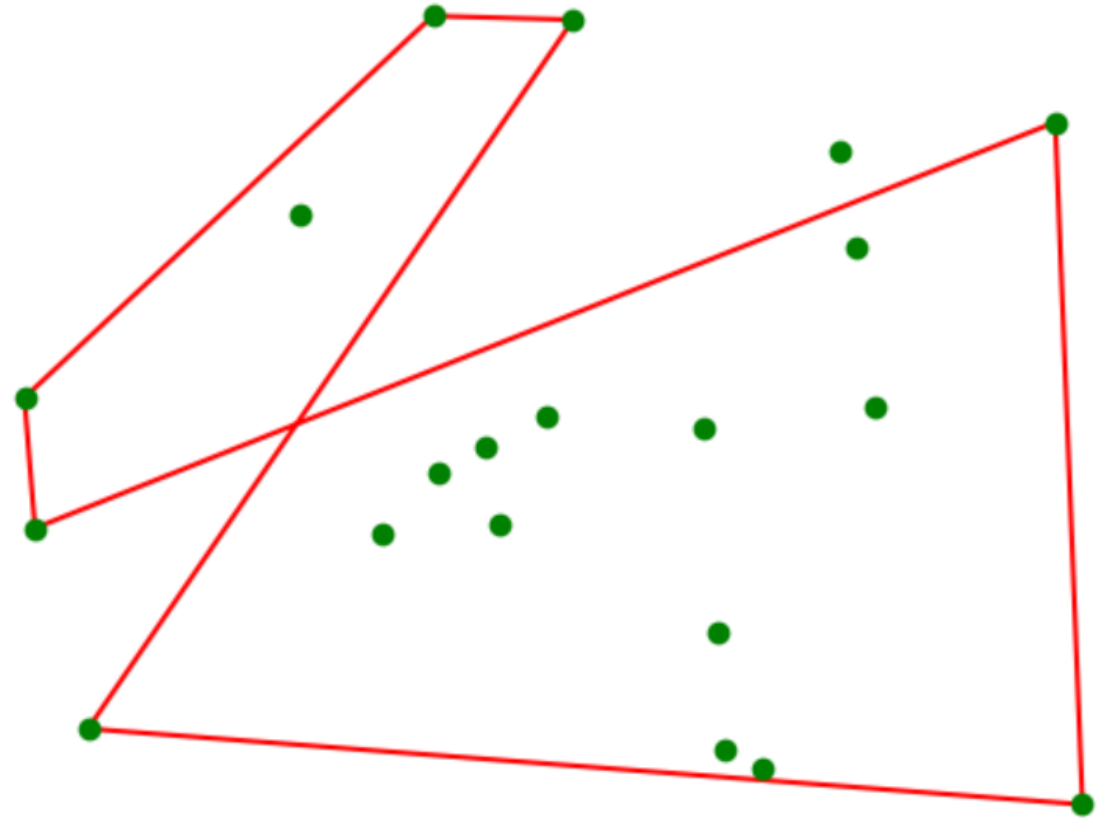
# Testing – how ?

- 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.
```

# pytest

- Run all tests stored in functions prefixed by `test` or `test_` prefixed test functions or methods inside Test prefixed test classes
- `pip install pytest`
- Run the `pytest` program from a shell

[pytest.org](https://pytest.org)

## binary-search-pytest.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

def test_binary_search():
    assert binary_search(42, []) == -1
    assert binary_search(42, [7]) == 0
    assert binary_search(42, [7,7,7,56,81]) == 2
    assert binary_search(8, [1,3,5,7,9]) == 3

import pytest

def test_types():
    with pytest.raises(TypeError):
        _ = binary_search(5, ['a', 'b', 'c'])
```

## Shell

```
> pytest binary-search-pytest.py
| ===== test session starts =====
| platform win32 -- Python 3.7.2, pytest-4.3.1, ... 0.9.0
| collected 2 items
| binary-search-pytest.py .. [100%]
| ===== 2 passed in 0.06 seconds =====
```

# 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'])

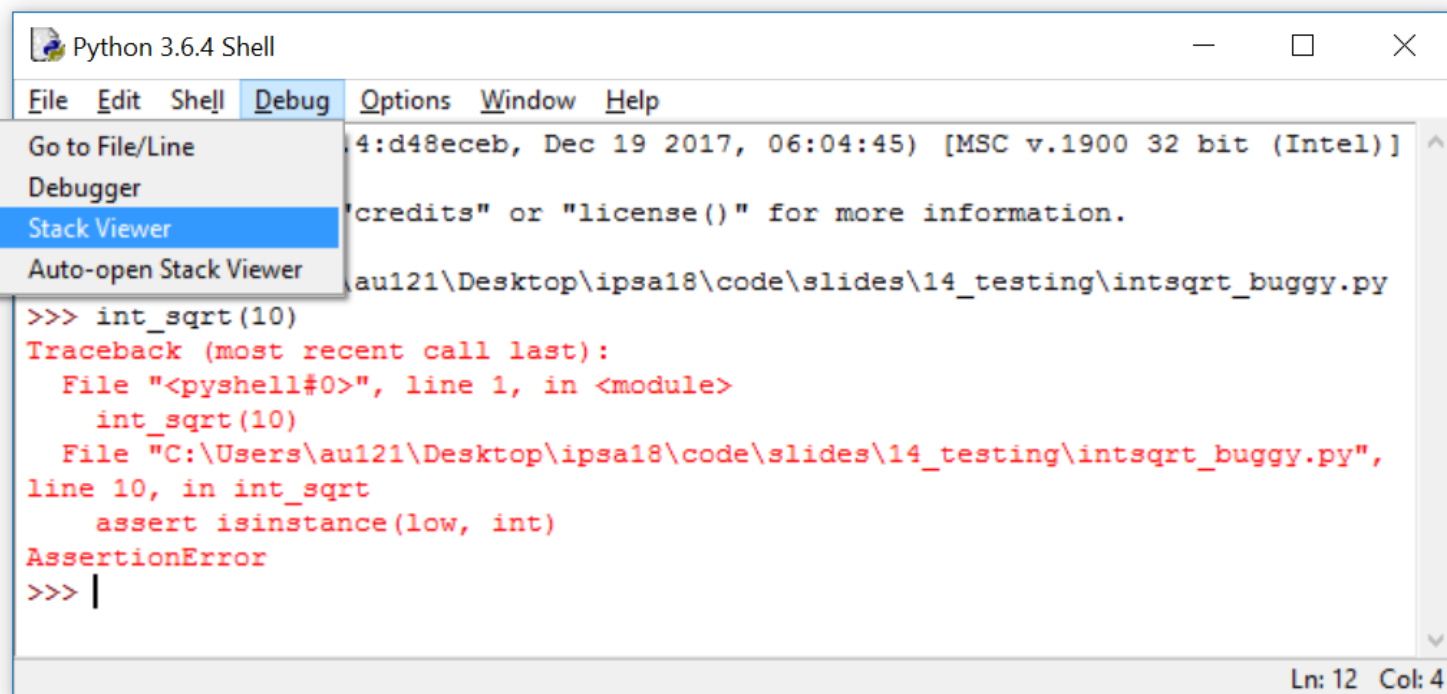
unittest.main(verbosity=2)
```

```
Python shell
```

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

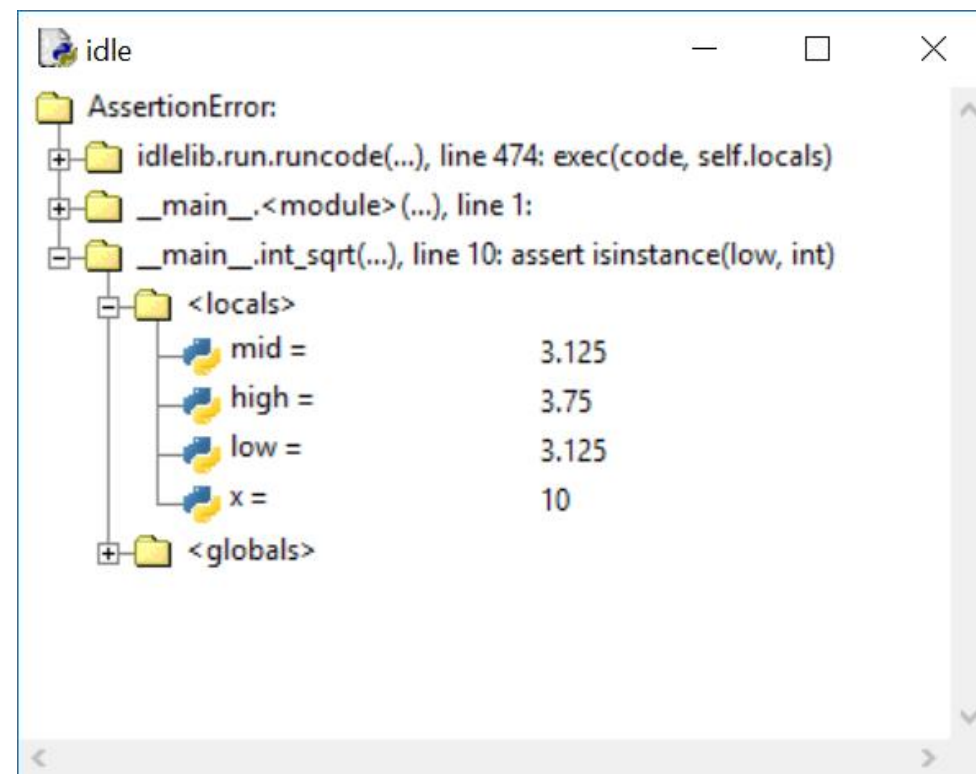
# Debugger (IDLE)

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



The screenshot shows the Python 3.6.4 Shell window. The 'Debug' menu is open, with 'Stack Viewer' selected. The shell displays a traceback for an AssertionError raised in the file 'intsqrt\_buggy.py' at line 10. The code being executed is 'int\_sqrt(10)'. The traceback shows the call stack starting from the Python shell, through the module, to the function 'int\_sqrt' where the error occurred. The error message is 'AssertionError'.

```
Python 3.6.4 Shell
File Edit Shell Debug Options Window Help
Go to File/Line
Debugger
Stack Viewer
Auto-open Stack Viewer
4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)]
credits" or "license()" for more information.
au121\Desktop\ipsa18\code\slides\14_testing\intsqrt_buggy.py
>>> int_sqrt(10)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    int_sqrt(10)
  File "C:\Users\au121\Desktop\ipsa18\code\slides\14_testing\intsqrt_buggy.py",
line 10, in int_sqrt
    assert isinstance(low, int)
AssertionError
>>> |
Ln: 12 Col: 4
```



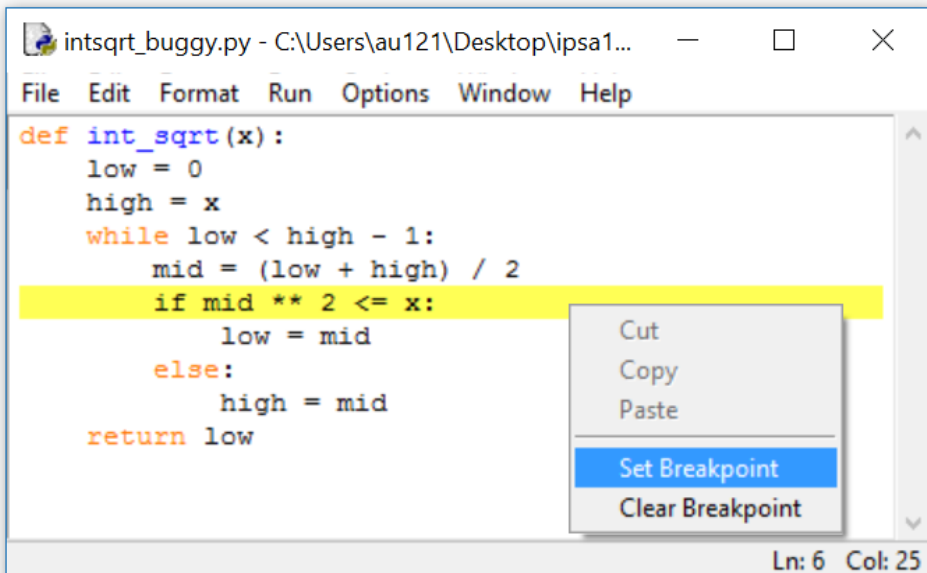
The screenshot shows the IDLE Stack Viewer window. It displays the call stack for the current exception. The stack starts with 'idlelib.run.runcode(...)' and goes down to the current frame 'int\_sqrt'. The local variables for the current frame are listed as follows:

Variable	Value
mid	3.125
high	3.75
low	3.125
x	10

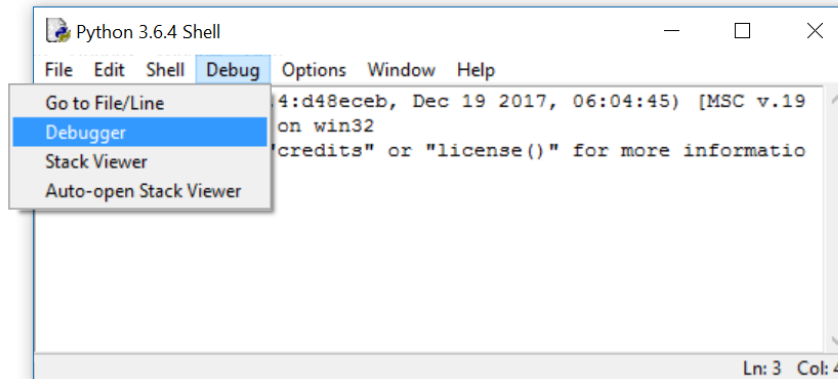


# Stepping through a program (IDLE debugger)

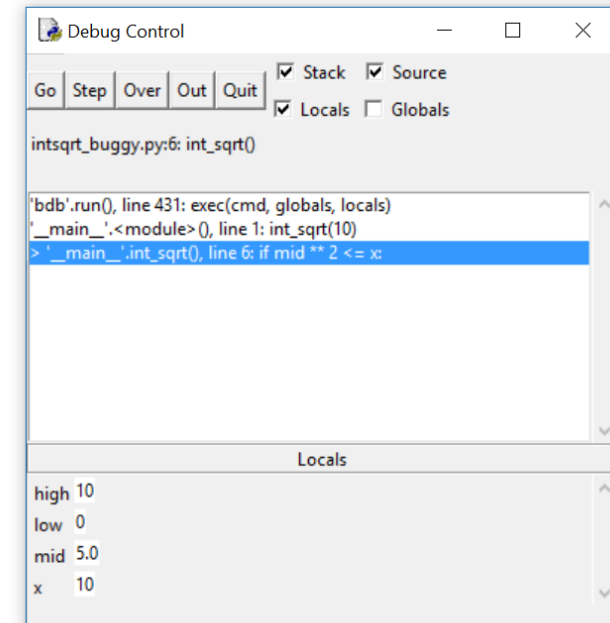
- **Debug > Debugger** in the Python shell opens Debug Control window
- **Right click** on a code line in editor to set a “breakpoint” in your code
- **Debug Control: Go** → run until next breakpoint is encountered;  
**Step** → execute one line of code; **Over** → run function call without details;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



```
intsqrt_buggy.py - C:\Users\au121\Desktop\ipsa1...
File Edit Format Run Options Window Help
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
Ln: 6 Col: 25
```



```
Python 3.6.4 Shell
File Edit Shell Debug Options Window Help
4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.19
on win32
credits" or "license()" for more informatio
Ln: 3 Col: 4
```



```
Debug Control
Go Step Over Out Quit [x] Stack [x] Source
[x] Locals [ ] Globals
intsqrt_buggy.py:6: int_sqrt()
'bdb'.run(), line 431: exec(cmd, globals, locals)
'_main_'.<module>, line 1: int_sqrt(10)
> '_main_'.int_sqrt(), line 6: if mid ** 2 <= x:
Locals
high 10
low 0
mid 5.0
x 10
```

# 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, ...**

# Mypy – a static type checker for Python

Experimental 

- **Static type checking** tries to analyze a program for potential type errors **without** executing the program
- Installing:  

```
pip install mypy
```
- Running Python will cause an error during execution, whereas using **mypy** the error will be found without executing the program
- Standard (and required) in statically typed languages like Java, C, C++

```
mypy-simple.py
```

```
print("start")  
print(42 + "abc") # error  
print("end")
```

```
Shell
```

```
> python mypy-simple.py  
| start  
| TypeError: unsupported operand type(s)  
| for +: 'int' and 'str'  
> mypy mypy-simple.py  
| mypy-simple.py:2: error: Unsupported  
| operand types for + ("int" and "str")
```

# Type hints (PEP 484)

- Python allows type hints in programs
- They are **ignored** at run-time by Python, but useful for static type analysis (mypy)
- Syntax

*variable* : *type*

*variable* : *type* = value

## mypy-basic-types.py

```
x : int # type hint
x = 42
x = "abc" # type error
y : int = 42 # type hint
y = "abc" # type error
z = 42
z = "abc" # type changed from int to str
print(x, y, z)
```

## Shell

```
> python mypy-basic-types.py
| abc abc abc
> mypy mypy-basic-types.py
| mypy-basic-types.py:3: error: Incompatible
| types in assignment (expression has type
| "str", variable has type "int")
| mypy-basic-types.py:6: error: ...
| mypy-basic-types.py:9: error: ...
```

# Type hints – functions

## Syntax

```
def name(variable : type, ...) -> return type
```

mypy-function.py	Shell
<pre>def f(x: int, units: str) -&gt; str:     return str(x) + " " + units  def g(x, units: str) -&gt; str:     return str(x) + " " + units  print(f(3, "cm")) print(f("one", "meter")) print(g(3, "cm")) print(g("one", "meter"))</pre>	<pre>&gt; python mypy-function.py   3 cm   one meter   3 cm   one meter &gt; mypy mypy-function.py   mypy-function.py:8: error: Argument 1   to "f" has incompatible type "str";   expected "int"</pre>

- Note: for functions and methods `function.__annotations__` is a dictionary with the annotation

# Type hints – objects

`mypy-classes.py`

```
class A:
    pass

class B(A):
    pass

class C:
    pass

a : A
b : B
c : C
```

```
a = A()
a = B() # valid, B subclass of A
a = C() # error
b = A() # error
b = B()
b = C() # error
c = A() # error
c = B() # error
c = C()
```

Shell

```
> mypy mypy-classes.py
| mypy-classes.py:15: error: Incompatible types in assignment (expression has type "C", variable has type "A")
| mypy-classes.py:16: error: Incompatible types in assignment (expression has type "A", variable has type "B")
| mypy-classes.py:18: error: Incompatible types in assignment (expression has type "C", variable has type "B")
| mypy-classes.py:19: error: Incompatible types in assignment (expression has type "A", variable has type "C")
| mypy-classes.py:20: error: Incompatible types in assignment (expression has type "B", variable has type "C")
```

# More type hints... see PEP 484 for even more...

```
mypy-typing.py
```

```
from typing import Mapping, Set, List, Tuple, Union, Optional

S : Set = {} # error {} dictionary
S2 : Set[int] = {1, 2, "abc"} # error "abc" is not int
D : Mapping[int, int] = {1: 42, 'a': 1} # error 'a' is not int
T : Tuple[int, str] = (42, 7) # error 7 is not str
L : List[Union[int, str]] = [42, 'a', None] # list can only contain int and str
L2 : List[Optional[str]] = ['abc', None, 42] # list can only contain str og None
```

```
Shell
```

```
> mypy mypy-function.py
| mypy-typing.py:3: error: Incompatible types in assignment (expression has type "Dict[<nothing>, <nothing>]", variable has type "Set[Any]")
| mypy-typing.py:4: error: Argument 3 to <set> has incompatible type "str"; expected "int"
| mypy-typing.py:5: error: Dict entry 1 has incompatible type "str": "int"; expected "int": "int"
| mypy-typing.py:6: error: Incompatible types in assignment (expression has type "Tuple[int, int]", variable has type "Tuple[int, str]")
| mypy-typing.py:7: error: List item 2 has incompatible type "None"; expected "Union[int, str]"
| mypy-typing.py:8: error: List item 2 has incompatible type "int"; expected "Optional[str]"
```