CSCI2202 Lecture 6: Exceptions and I/O

Finlay Maguire (finlay.maguire@dal.ca)

TAs: Ehsan Baratnezhad (ethan.b@dal.ca); Precious Osadebamwen (precious.osadebamwen@dal.ca)

Overview

- Exceptions
- Filesystems
- Reading/Writing Files
- Assertions & Testing
- Practice Questions

This lecture draws heavily on Gerth Stølting Brodal's Excellent IPSA Course at Aarhaus University

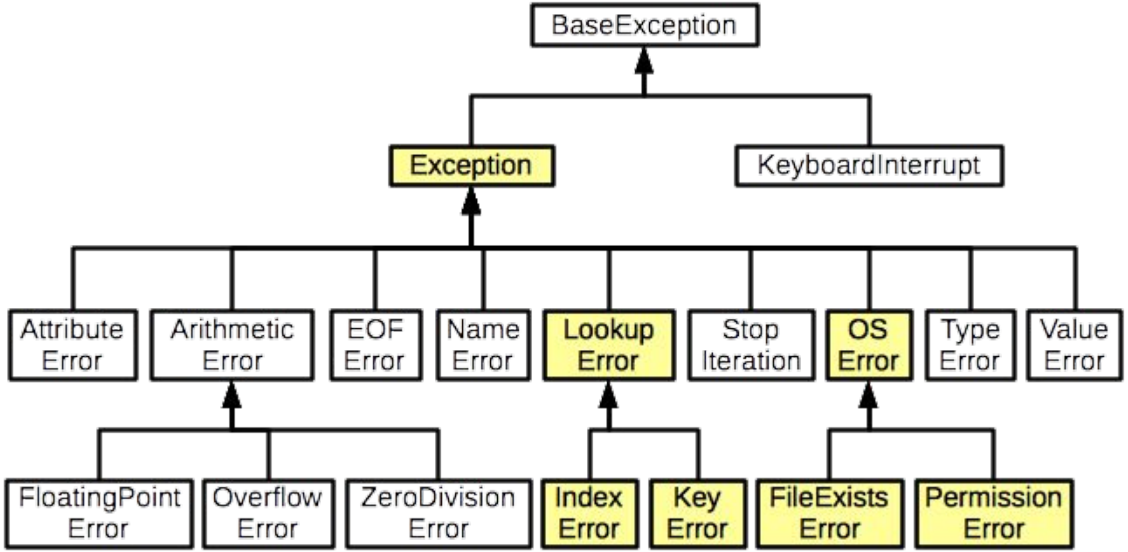
Exceptions

Exceptions are the the errors than python “throws”/”raises”

```
>>> x = [1,2,3]

>>> x[10]

IndexError: list index out of range

>>> y = {'a': 10, 'b': 50}

>>> y[c]

KeyError: 'c'

>>> '155' + []

TypeError: can only concatenate str (not
"list") to str

>>> import fake_module

ModuleNotFoundError: No module named
'fake_module'
```

```
>>> z

NameError: name 'z' is not defined

>>> number = 42

>>> number.append(1)  #

AttributeError: 'int' object has no
attribute 'append'

>>> int('hello')

ValueError: invalid literal for int() with
base 10: 'Hello'

>>> if x = 5

SyntaxError: invalid syntax. Maybe you
meant '==' or ':=' instead of '='?
```

Exceptions exist in a hierarchy



Big 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

We handle exceptions ourselves by `catch`-ing them

```
def divide(x,y):

      return x / y

y = divide(5, 0)

ZeroDivisionError: division by zero # crash


y = Divide(10, ‘12x’)

TypeError: unsupported operand type(s) for /:
'int' and 'str'
```

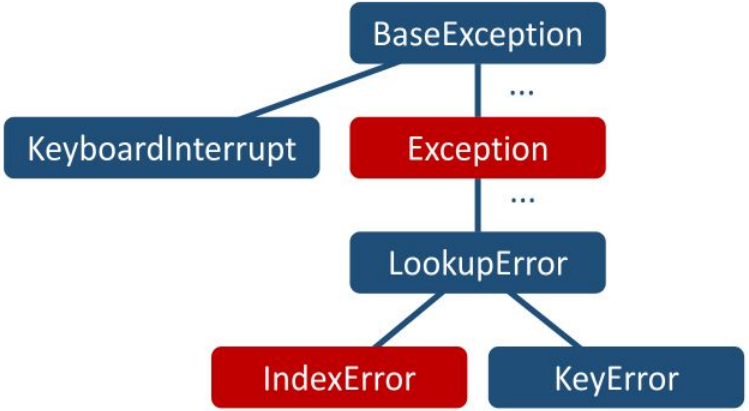
```
def divide(x,y):

      try:

            return x / y

      except ZeroDivisionError:

            print("Can’t divide by zero")

      return

y = divide(5, 0) # caught ZeroDivisonError

"Can’t divide by zero"

y = divide(10, ‘12x’)

TypeError: unsupported operand type(s) for /:
'int' and 'str'
```

Can catch multiple different exceptions

```
def divide(x,y):

    try:

        return x / y

    except ZeroDivisionError:

        print("Can’t divide by zero, returning None")

    except TypeError:

        print(“Invalid types, returning None”)

    # return
```

```
y = divide(5, 0) # y==None

“Can’t divide by zero, returning None”

# caught the ZeroDivisonError

y = divide(10, ‘12x’)

“Invalid types, returning None”

TypeError: unsupported operand type(s) for /:
'int' and 'str'
```

Hierarchy means order of these except statements matter



except-twice1.py

```python
try:
    L[4]
except IndexError:  # must be before Exception
    print('IndexError')
except Exception:
    print('Fall back exception handler')
```

except-twice2.py

```python
try:
    L[4]
except Exception:  # and subclasses of Exception
    print('Fall back exception handler')
except IndexError:
    print('IndexError')  # unreachable
```

`try` statement syntax

arbitrary number of except cases

```
try:
    code
except ExceptionType1:
    code  # executed if raised exception instanceof
          # ExceptionType1 (or subclass of ExceptionType1)
except ExceptionType2:
    code  # executed if exception type matches and none of
          # the previous except statements matched
...
else:
    code  # only executed if no exception was raised
finally:
    code  # always executed independent of exceptions
          # typically used to clean up (like closing files)
```

try, except, else, finally

```
def divide_numbers(a, b):
    try:
        result = float(a) / float(b)
    except ZeroDivisionError:
        print(“Can’t divide by zero”)
    except ValueError:
        print(f“Can’t convert {a} or {b} to floats”)
    else:
        return result
    finally:
        print(“Calculation complete”)
```

```
divide_numbers(‘a’, 0) # returns None
“Can’t convert ‘a’ or 0 to floats”
“Calculation complete”

divide_numbers(1, 0) # returns None
“Can’t divide by zero”
“Calculation complete”

divide_numbers(1, 2) # returns 0.5
“Calculation complete”
```

`except` variations

```
except:                    # catch all exceptions

except ExceptionType:      # only catch exceptions of class ExceptionType
                           # or subclasses of ExceptionType

except (ExceptionType1, ExceptionType2, ..., ExceptionTypek):
                           # catch any of k classes (and subclasses)
                           # paranthesis cannot be omitted

except ExceptionType as e:
                           # catch exception and assign exception object to e
                           # e.args contains arguments to the raised exception
```

Raising exceptions

- An exception is raised (or trown) using one of the following (the first being an alias for the second):

```
raise ExceptionType
raise ExceptionType()
raise ExceptionType(args)
```

abstract.py

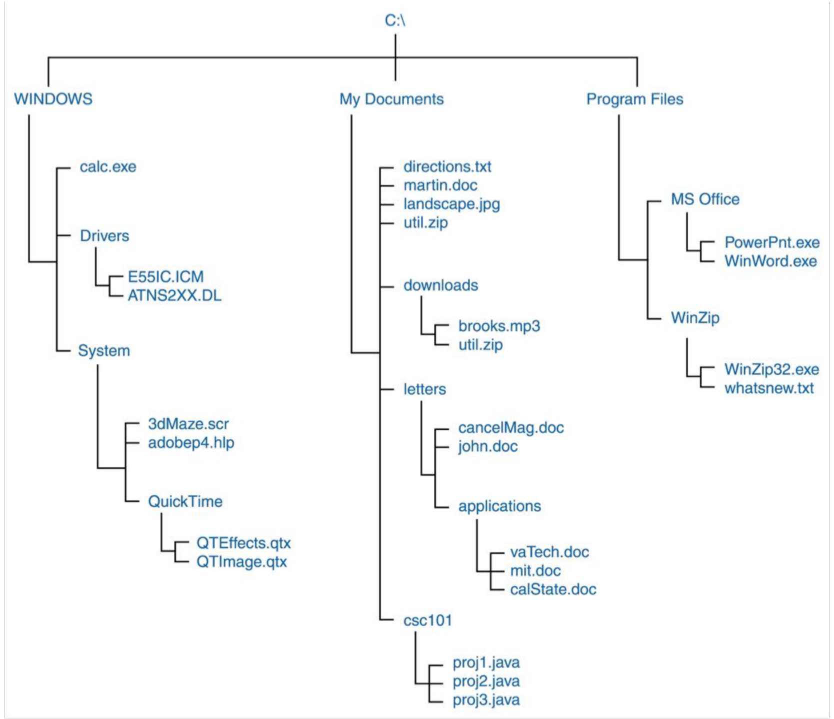
```
class A():
      def f(self):
          print('f')
          self.g()
      def g(self):
          raise NotImplementedError
class B(A):
      def g(self):
          print('g')
```

Python shell

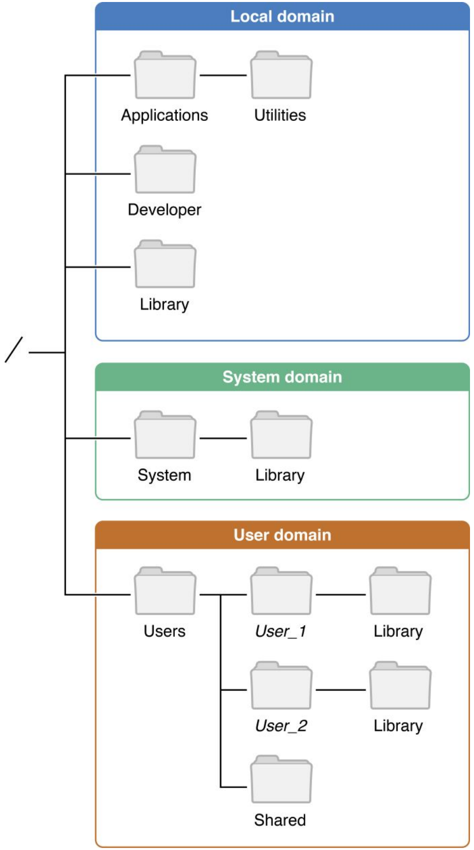
```
> B().f()
| f
| g
> A().f()
| f
| NotImplementedError
```

Dealing with files

What is a filesystem?

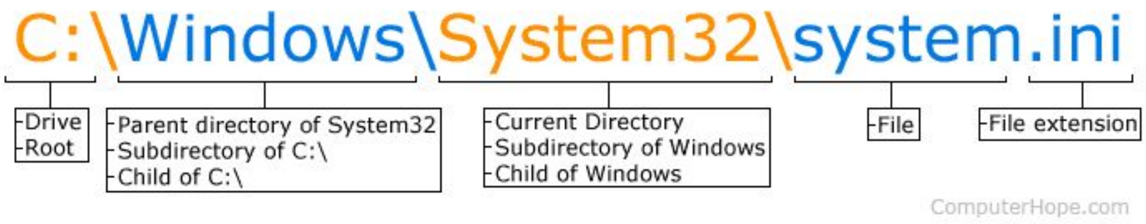


https://w3.cs.jmu.edu/spragunr/jmu_unix_tutorial/unixintro.html



https://developer.apple.com/library/archive/documentation/FileManagement/Conceptual/FileSystemProgrammingGuide/FileSystemOverview/FileSystemOverview.html

Parent and child folders



Windows: C:\Users\smith\Documents\School\CSCI2202\lab6.ipynb

Mac: /Users/smith/Documents/School/CSCI2202/lab6.ipynb

Moving up and down folder hierarchies - relative paths

- CSCI2202
    - Lab4
        - filez.txt
    - Lab5
        - lab_notebook5.ipynb
    - **Lab6**
        - lab_notebook6.ipynb
        - input_files
            - file1.txt
            - file2.txt

Assuming we started jupyter in lab6 folder

```
From pathlib import Path

Path("../CSCI2202").exists()
True

Path("blah").exists()
False

Path("../CSCI2202/Lab4").is_dir()
True
Path("../CSCI2202/Lab4").is_file()
False

Path("../CSCI2202/Lab5/lab_notebook6.ipynb").is_dir()
False
Path("../CSCI2202/Lab5/lab_notebook6.ipynb").is_file()
True
```

Printing contents of a folder

- CSCI2202
  - Lab4
    - filez.txt
  - Lab5
    - lab_notebook5.ipynb
  - **Lab6**
    - lab_notebook6.ipynb
    - input_files
      - file1.txt
      - file2.txt

Assuming we started jupyter in lab6 folder

```
for item in Path("CSCI2202").iterdir():
    print(item)
../CSCI2202/Lab4
../CSCI2202/Lab5
../CSCI2202/Lab6


for item in Path("CSCI2202/Lab5").iterdir():
    print(item)
../CSCI2202/Lab5/lab_notebook5.ipynb
```

3 ways to read lines from a file

Steps

1. Open file using `open`
2. Read lines from file using
   a) `for line in filehandler:`
   b) `filehandler.readlines`
   c) `filehandler.readline`
3. Close file using `close`

`open ('filename.txt')` assumes the file to be in the same folder as your Python program, but you can also provide a full path `open('c:/Users/gerth/Documents/filename.txt')`

try to open file for reading

filename

filehandle

iterate over lines in file

close file when done

reading-file1.py

```
f = open('reading-file1.py')
for line in f:
    print('> ', line, end='')
f.close()
```

read all lines into a list of strings

reading-file2.py

```
f = open('reading-file2.py')
lines = f.readlines()
f.close()
for line in lines:
    print('> ', line, end='')
```

read single line (terminated by '\n')

reading-file3.py

```
f = open('reading-file3.py')
line = f.readline()
while line != '':
    print('> ', line, end='')
    line = f.readline()
f.close()
```

3 ways to write lines to a file

- Opening file:
  `open(filename, mode)`
  where *mode* is a string, either `'w'` for opening a new (or truncating an existing file) and `'a'` for appending to an existing file
- Write single string:
  `filehandle.write(string)`
  Returns the number of characters written
- Write list of strings strings:
  `filehandle.writeline(list)`
- Newlines (`'\n'`) must be written explicitly
- `print` can take an optional `file` argument

try to open file for writing

write mode

write single string to file

write list of strings to file

append to existing file

write-file.py

```
f = open('output-file.txt', 'w')
f.write('Text 1\n')
f.writelines(['Text 2\n', 'Text 3 '])
f.close()

g = open('output-file.txt', 'a')
print('Text 4', file=g)
g.writelines(['Text 5 ', 'Text 6'])
g.close()
```

output-file.txt

```
Text 1
Text 2
Text 3 Text 4
Text 5 Text 6
```

Exceptions while dealing with files

- When dealing with files one should be prepared to handle errors / raised exceptions, e.g. `FileNotFoundError`

reading-file4.py

```
try:
    f = open('reading-file4.py')
except FileNotFoundError:
    print('Could not open file')
else:
    try:
        for line in f:
            print('> ', line, end='')
    finally:
        f.close()
```

Opening files using **`with`** (recommended way)

- The Python keyword `with` allows to create a *context manager* for handling files
- Filehandle will automatically be closed, also when exceptions occur
- Under the hood: filehandles returned by `open` support `__enter__` and `__exit__` methods

f = result of calling `__enter__()`
on result of `open` expression,
which is the file handle

reading-file5.py

```
with open('reading-file5.py') as f:
    for line in f:
        print('> ', line, end='')
```

docs.python.org/3/library/stdtypes.html#typecontextmanager

Checking if a file exists/manipulating filepaths

```
>>> from pathlib import Path

>>> x = Path("fake_file.txt")

>>> x.exists()

False

# create fake_file.txt in the parent
folder

>>> x = Path("../fake_file.txt")

>>> x.exists()

True
```

Note: Examples online may use `os.path` for this type of functionality but this is outdated

```
# create new childfolder/subfolder called
"test" containing  fake_file2.txt

>>> x = Path("test/fake_file.txt")

>>> x.exists()

True


if Path("file.txt").exists():

    print("file exists")

else:

    print("file doesn't exist")
```

Performance of scanning a file

- Python can efficiently scan through quite big files

File	Size	Time
Atom_chem_shift.csv	≈ 750 MB	≈ 8 sec
cano.txt	≈ 3.7 MB	≈ 0.1 sec

The first search finds all lines related to ThrB12-DKP-insulin (Entry ID 6203) in a chemical database available from www.bmrb.wisc.edu

The second search finds all occurrences of "Germany" in Conan Doyle's complete Sherlock Holmes available at sherlock-holm.es

file-scanning.py

```python
from time import time

for filename, query in [
        ('Atom_chem_shift.csv', ',6203,'),
        ('cano.txt', 'Germany')
    ]:
    count = 0
    matches = []
    start = time()
    with open(filename) as f:
        for i, line in enumerate(f, start=1):
            count += 1
            if query in line:
                matches.append((i, line))
    end = time()

    for i, line in matches:
        print(i, ':', line, end='')
    print('Duration:', end - start)
    print(len(matches), 'of', count, 'lines match')
```

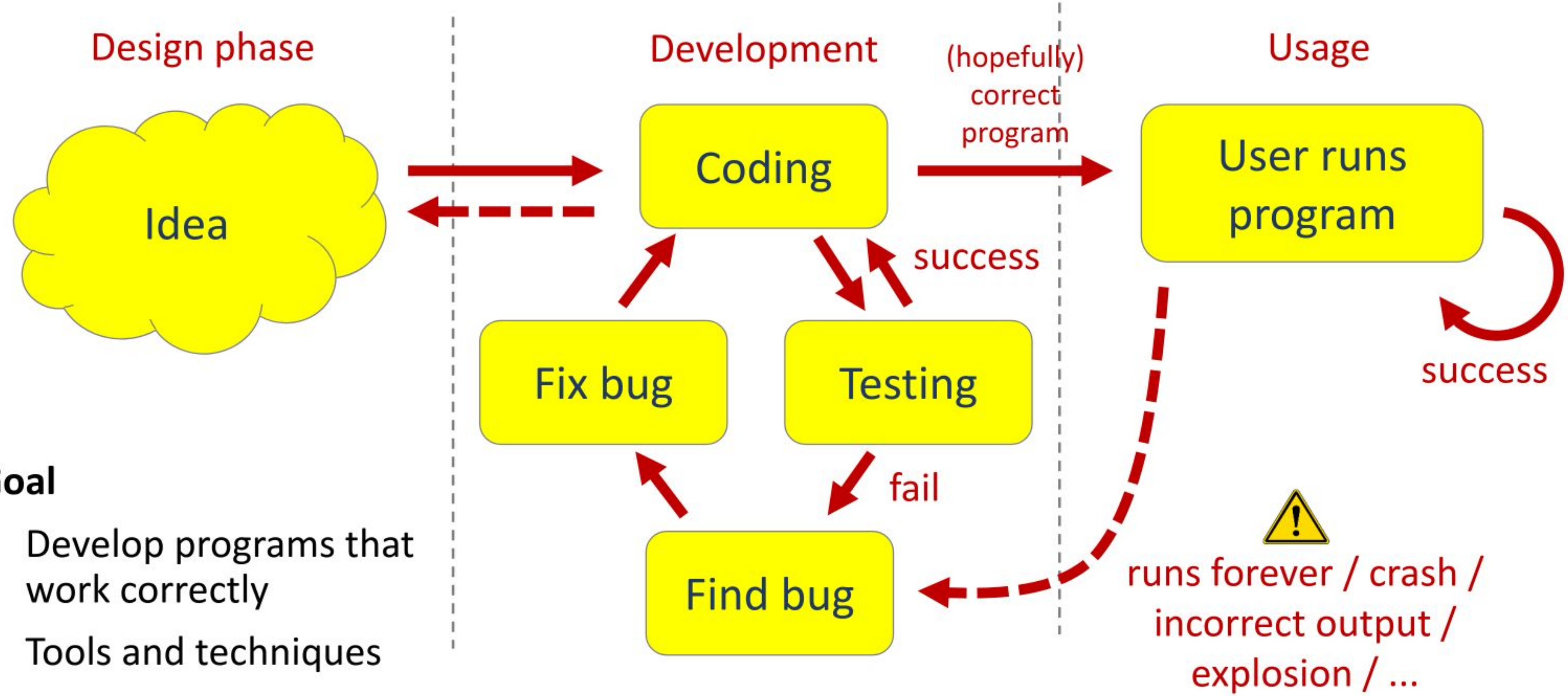
Python shell

```
...
3057752 : 195,,2,2,30,30,THR,HB,H,1,4.22,0.02,,1,,,,,,,,,,,,,,,,,,,,,228896,6203,2
3057753 : 196,,2,2,30,30,THR,HG21,H,1,1.18,0.02,,1,,,,,,,,,,,,,,,,,,,,,228896,6203,2
3057754 : 197,,2,2,30,30,THR,HG22,H,1,1.18,0.02,,1,,,,,,,,,,,,,,,,,,,,,228896,6203,2
3057755 : 198,,2,2,30,30,THR,HG23,H,1,1.18,0.02,,1,,,,,,,,,,,,,,,,,,,,,228896,6203,2
Duration: 7.760039329528809
329 of 9758361 lines match
57557 :       "Well, then, to the West, or to England, or to Germany, where father
66515 :       kind master. He wanted me to go with his wife to Germany yesterday,
66642 :       of business in Germany in the past and my name is probably familiar
73273 :       associates with Germany. This he placed in his instrument cupboard.
Duration: 0.07700657844543457
4 of 76764 lines match
```

Writing good code!

- On average, a developer creates 70 bugs per 1000 lines of code
- 15 bugs per 1,000 lines of code find their way to the customers
- Fixing a bug takes 30 times longer than writing a line of code
- 75% of a developer’s time is spent on debugging

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...

What are the steps to achieving this?

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” – Dijkstra

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)      raises SystemExit

f(0)
```

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

```python
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

```python
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
- try to fail fast to discover errors early – making debugging easier

infinite-recursion6.py

```python
def f(depth):
    if not depth <= 100:
        raise AssertionError('runaway recursion???')
    f(depth + 1)

f(0)
```

Python shell

```
| File "...\infinite-recursion6.py", line 3, in f
|     raise AssertionError("runaway recursion???")
| AssertionError: runaway recursion???
```

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.

    Requires x >= 0 is an integer.
    Returns the 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.
|
|     Requires x >= 0 is an integer.
|     Returns the 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.

    Requires x >= 0 is an integer.
    Returns the integer floor(sqrt(x)).'''

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

check input 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.

    Requires x >= 0 is an integer.
    Returns the 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 function specifications are satisfied 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:  # low <= floor(sqrt(x)) < high
    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

$\lfloor\sqrt{x}\rfloor \in$ [low, high[

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.

    Requires x >= 0 is an integer.
    Returns the integer floor(sqrt(x)).'''

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

    low, high = 0, x + 1
    while low < high - 1:  # low <= floor(sqrt(x)) < high
        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
```

Systematic Testing

Test driven development / Stress tests / Random testing

- Test driven development
  Write the tests before functionality – only write code needed by tests
- The challenge – what tests to do?
  Can you manually find all relevant cases? In particular all edge cases?
- Automate the testing?
  - Write method that can verify the output (possibly slower than the method)
  - Systematically try *all* possible inputs (if range is small)
  - Try a large random subset of inputs (if many possible inputs)

intsqrt_automatic_testing.py

```python
import random

def int_sqrt(x):
    return 42  # Dummy code - write test code first

def test_int_sqrt(x):
    print('.', end='', flush=True)  # Show progress
    assert x >= 0  # Verify input
    answer = int_sqrt(x)
    # Verify output
    assert answer ** 2 <= x < (answer + 1) ** 2

# Test small inputs
for x in range(0, 100):
    test_int_sqrt(x)

# Test increasing sized inputs
for d in range(3, 30):
    for _ in range(100):  # Repeat for each size
        test_int_sqrt(random.randint(1, 10 ** d))
```

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

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

```python
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

Overview of testing

- 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, ...**

Common Practical Issues

Indexing

What is `[7,3,5][[1,2,3][1]]`?

a) 1

b) 2

c) 3

d) 5

e) 7

Indexing

What is `[7,3,5][[1,2,3][1]]`**?**

a) 1

b) 2

c) 3

d) 5 # [1,2,3][1] == 2; [7,3,5][2] == 5

e) 7

Aliasing/Copying

```
a = [[3,5],[7,11]]
b = a
c = a[:]
a[0][1] = 4
c[1] = b[0]
```

What is c?

a) [[3,5],[7,11]]

b) [[3,5],[3,5]]

c) [[3,4],[3,5]]

d) [[3,4],[3,4]]

Aliasing/Copying

```
a = [[3,5],[7,11]]
b = a
c = a[:]
a[0][1] = 4
c[1] = b[0]
```

What is c?

a) [[3,5],[7,11]]

b) [[3,5],[3,5]]

c) [[3,4],[3,5]]

d) [[3,4],[3,4]]

`a = [[3,5],[7,11]]`

- Creates a nested list `a` with two inner lists: `[3,5]` and `[7,11]`

`b = a`

- Creates a reference to the same list - `b` and `a` point to the exact same object in memory
- This is called a shallow copy, or more accurately, just creating another reference

`c = a[:]`

- Creates a shallow copy of list `a`
- The outer list is copied, but the inner lists are still references to the same objects
- This is different from `b = a` because `c` is a new list object

`a[0][1] = 4`

- Changes the second element of the first inner list from 5 to 4
- Because both `b` and `c` contain references to the same inner lists, this change affects all three variables

`c[1] = b[0]`

- Takes the first inner list of `b` (`[3,4]`) and assigns it to the second position of `c`

Objects 1

```
class A:
    def f(self):
        print(“Af”)
        self.g()
    def g(self):
        print(“Ag”)

class B(A)
    def g(self):
        print(“Bg”)

b = B()
b.f()
?
```

What does b.f() print?

a) AttributeError
b) Af Ag
c) Af Bg
d) None

Objects 1

```
class A:
    def f(self):
        print("Af")
        self.g()
    def g(self):
        print("Ag")

class B(A)
    def g(self):
        print("Bg")

b = B()
b.f()
?
```

What does b.f() print?

a) AttributeError
b) Af Ag
c) Af Bg
d) None

Objects 2

```
class MyClass:
    x = 2
    def get(self):
        self.x += 1
         return MyClass.x + self.x

c = MyClass()
c.get()
?
```

What does c.get() return?

a) 4
b) 5
c) 6
d) UnboundLocalError

Objects 2

```
class MyClass:
    x = 2
    def get(self):
        self.x += 1
         return MyClass.x + self.x

c = MyClass()
c.get()
?
```

What does c.get() return?

a) 4
b) 5
c) 6
d) UnboundLocalError