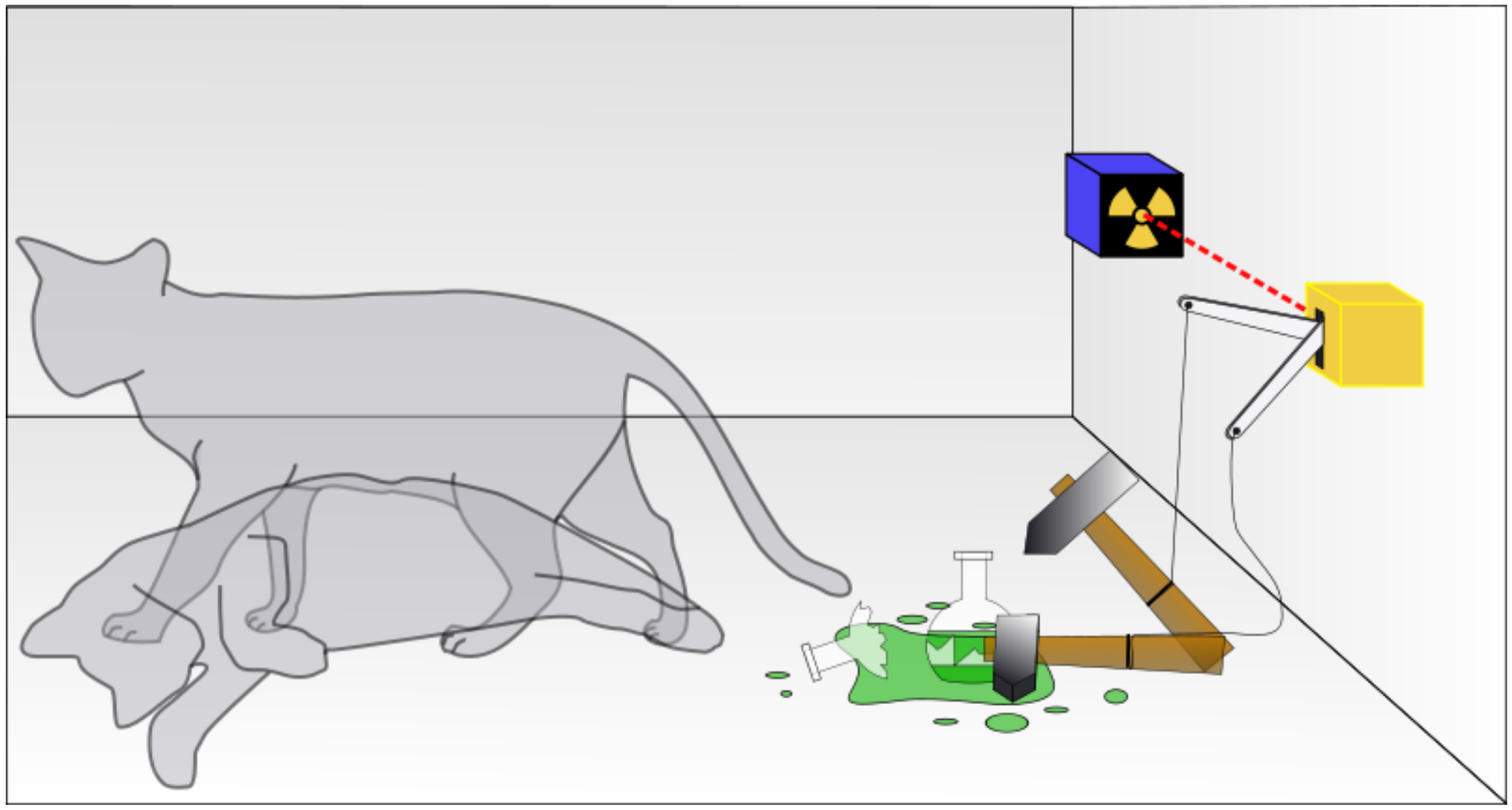
Best Programming Practices

Ashish Mahabal, CD3, Caltech
LSSDS
2018-08-27

Code Divides the Universe

Treat your coding accordingly



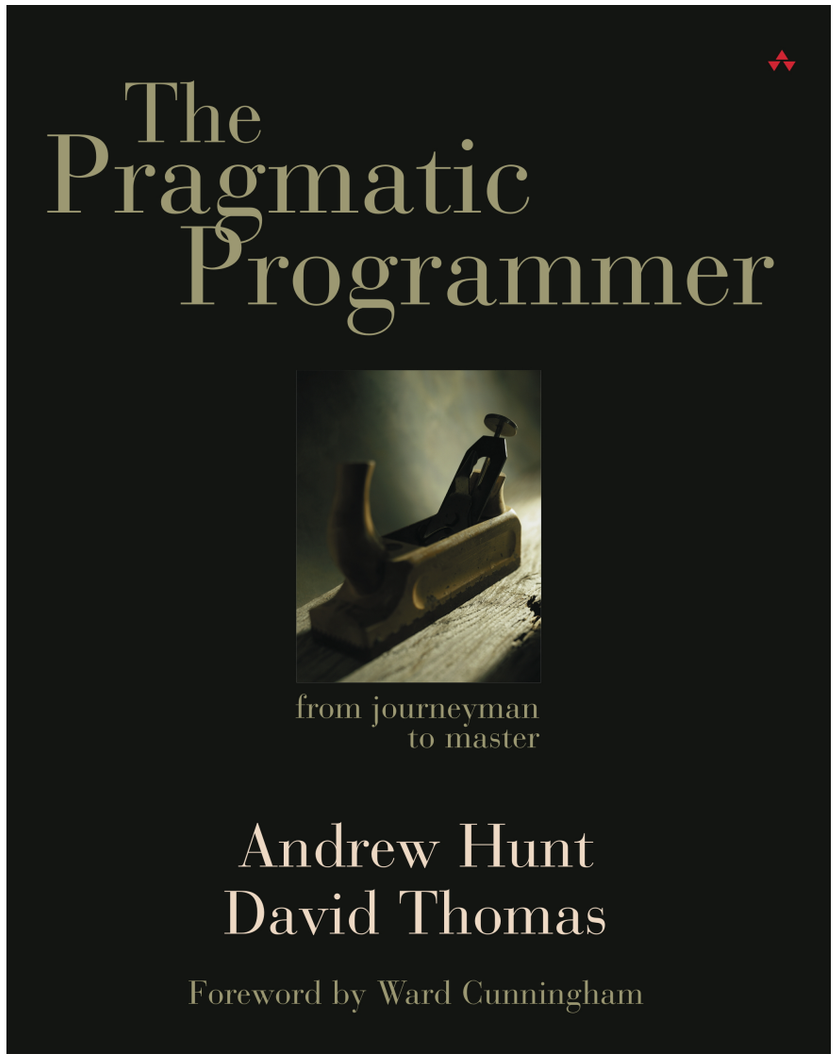
"Schrodingers cat" by Dhatfield – SA 3.0 (Wikimedia)

Vision of a program

- Variables
  - their names
- Subroutines
  - their names
  - their functions
- Structure of a program
- Evolution and well being

Only small variations based on tools

The Pragmatic Programmer
By Andrew Hunt and David Thomas



for Python:

When in doubt: import this

Source Code Control

Allow versions to be stored in one place

Allow multiple people to work on a piece of code

Allow access from multiple computers easily

[Concurrent Version Systems (CVS)]

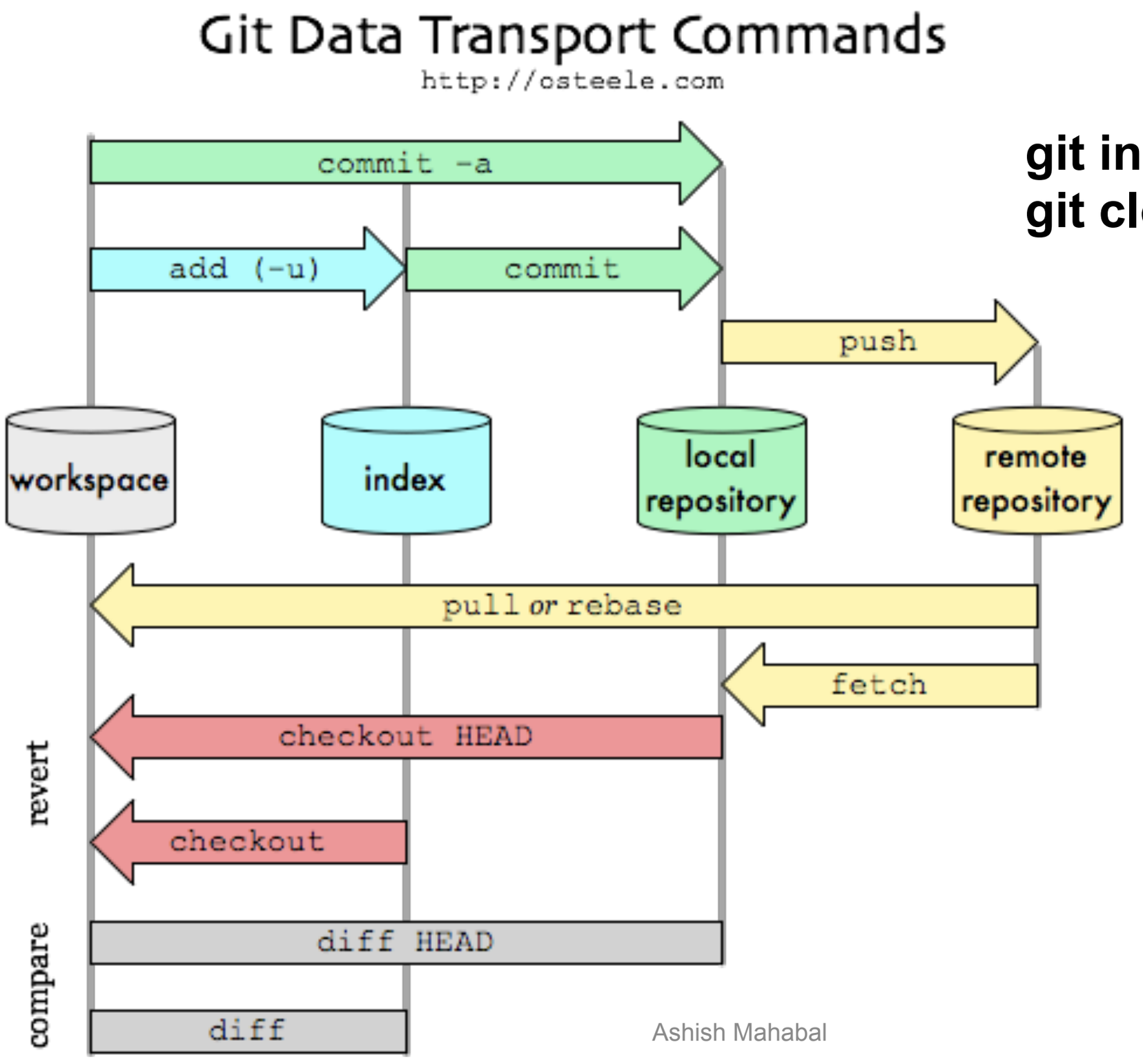
Apache SubVersion (SVN)

Git

[Mercurial]

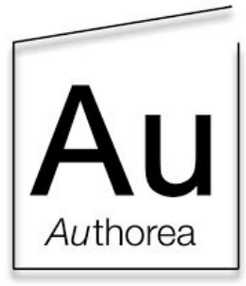
Git

git init
git clone repo



Online “hubs” that allow versioning

- github
- bitbucket
- google drive
- authorea – collaborative papers
- overleaf – collaborative latexing



Coding by instinct

- Variable names
  - UpperCamelCase,
  - lowerCamelCase,
  - alllowercase (bad!)
  - period.separated (issues with modules)
  - underscore_separated (possible issues with latex)

Coding by instinct (loops)

- Types of loops (for, while, …)

```
for <variable> in <sequence>:
    <statements>
else:
    <statements>
```

```
for k in {"x": 1, "y": 2}:
    print k
```

Even avoiding explicit loops …

Sum numbers between 1 and 100 that are divisible by 3 or 5 but not both

```
sum = 0

for n in range(1000):
    if (n % 3 == 0) or (n % 5 == 0):
        sum += n

print("Sum =", sum)
```

```
('Sum =', 233168)
```

```
import numpy as np
print(np.sum([ x for x in xrange(1000) if x%3==0 or x%5==0 ]))
```

```
233168
```

```
import numpy as np
np.sum(range(0, 1000, 3)) + np.sum(range(0, 1000, 5)) - np.sum(ra
```

```
233168
```

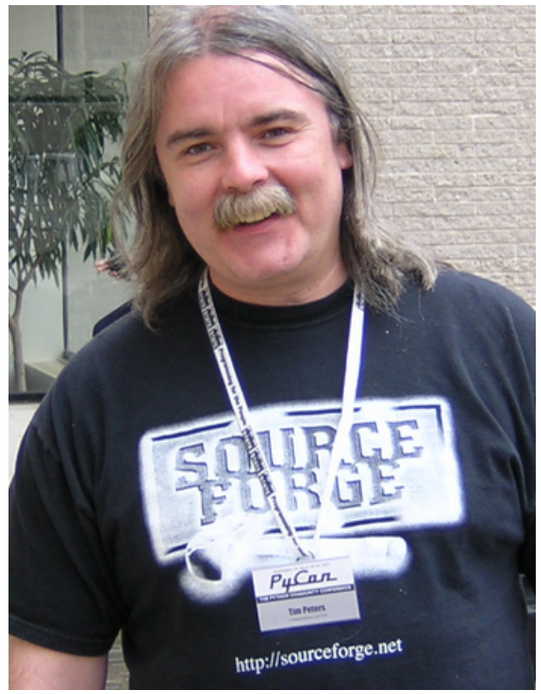
Coding by instinct

- Variable names
- Types of loops (for, while, …)
- Formatting
  - Indents, brackets, braces, semicolons
- Procedural versus object oriented approach

Conscious and consistent programming style

Zen of Python (PEP 20 – Aug 2004)

1. Beautiful is better than ugly.
2. Explicit is better than implicit.
3. Simple is better than complex.
4. Complex is better than complicated.
5. Flat is better than nested.
6. Sparse is better than dense.
7. Readability counts.
8. Special cases aren't special enough to break the rules.
9. Although practicality beats purity.
10. Errors should never pass silently.
11. Unless explicitly silenced.
12. In the face of ambiguity, refuse the temptation to guess.
13. There should be one— and preferably only one —obvious way to do it.
14. Although that way may not be obvious at first unless you're Dutch.
15. Now is better than never.
16. Although never is often better than *right* now.
17. If the implementation is hard to explain, it's a bad idea.
18. If the implementation is easy to explain, it may be a good idea.
19. Namespaces are one honking great idea — let's do more of those!



(a poem by Tim Peters)

Available as “import this”

Modification cycle

Write test

Run and make sure it fails

Checkout

Change, comment, edit readme etc.

Compile

Run: make sure test passes

Checkin

A simple test

```
#python -m unittest test_module1 test_module2

import unittest

def fun(x):
    return x + 1

class MyTest(unittest.TestCase):
    def test(self):
        self.assertEqual(fun(3), 4)
```

Before the project

Dig for requirements

Document requirements

Make use case diagrams

Maintain a glossary

Document, Document, …

Easy development versus easy maintenance

- projects live much longer than intended
- adopt more complex and readable language

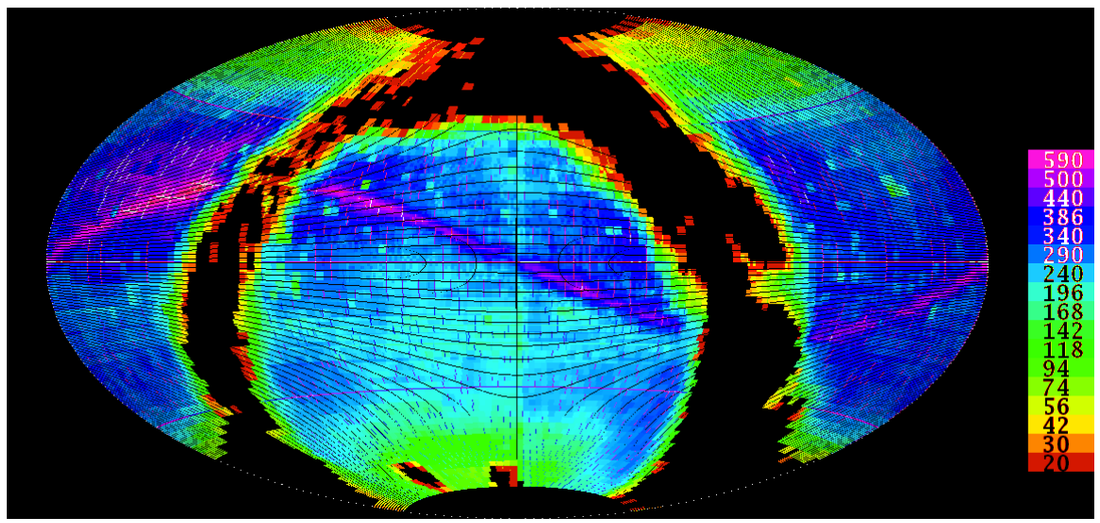
Check requirements

Design, implement, integrate

Validate

Validation

- Don’t trust the work of others
  - Validate data (numbers, chars etc.)
  - Put constraints (-90 <= dec <= 90)
  - Check consistency



Validation

- Don’t trust the work of others
  - Validate data
  - Put constraints
  - Check consistency
- Don’t trust yourself
  - Do all the above to your code too

When something goes wrong

- Crash early
  - Sqrt of negative numbers (require, ensure, NaN)
- Crash, don't trash
  - Die
  - Croak (blaming the caller)
  - Confess (more details)
  - Try/catch (own error handlers e.g. HTML 404)
- Exceptions – when to raise them
  - should it have existed?
  - Don't know?

try/except

Yes:

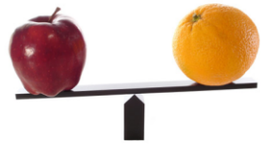
```
try:
    value = collection[key]
except KeyError:
    return key_not_found(key)
else:
    return handle_value(value)
```

No:

```
try:
    # Too broad!
    return handle_value(collection[key])
except KeyError:
    # Will also catch KeyError raised by handle_value()
    return key_not_found(key)
```

- Don’t optimize code – benchmark it
- Don’t optimize data structures – measure them
- Cache data when you can – use Memoize
- Benchmark caching strategies
- Don’t optimize applications – profile them (find where they spend most time)

factorial of k



gridgain.blogspot.com

Momoization

```
factorial_memo = {}
def factorial(k):
    if k < 2: return 1
    if not k in factorial_memo:
        factorial_memo[k] = k * factorial(k-1)
    return factorial_memo[k]

factorial(10)
```

Profiling

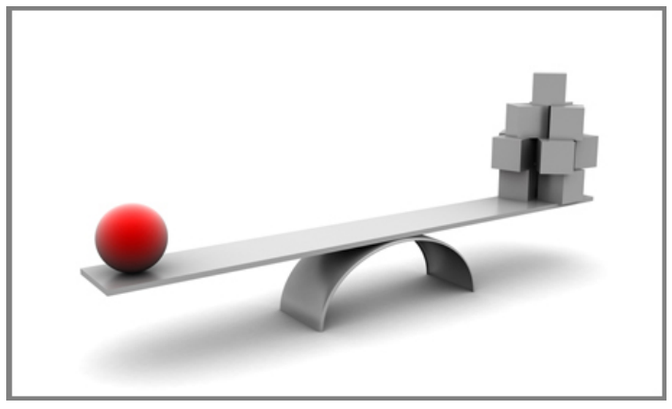
```
import cProfile
import re
cProfile.run('re.compile("Hello|World")')
```

```
         238 function calls (233 primitive calls) in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 re.py:188(compile)
        1    0.000    0.000    0.000    0.000 re.py:226(_compile)
        1    0.000    0.000    0.000    0.000 sre_compile.py:178(_compile_charset)
        1    0.000    0.000    0.000    0.000 sre_compile.py:207(_optimize_charset)
        4    0.000    0.000    0.000    0.000 sre_compile.py:24(_identityfunction)
      3/1    0.000    0.000    0.000    0.000 sre_compile.py:32(_compile)
        1    0.000    0.000    0.000    0.000 sre_compile.py:361(_compile_info)
        2    0.000    0.000    0.000    0.000 sre_compile.py:474(isstring)
```

Benchmarking

Benchmarking game:

http://shootout.alioth.debian.org/



blog.insresearch.com

Benchmarking python:

http://ziade.org/2007/10/18/unobtrusive-benchmark-and-debug-of-python-applications/

Necessary ingredients

- Robustness
- Efficiency
- Maintainability

Robustness

- Introducing (tests for) errors
  - checking for existence (uniform style)
- Edge cases
  - 0? 1? last?
- Error handling
  - exceptions? Verifying terminal input
- Reporting failure
  - Traces? Errors don’t get quietly ignored

Checking for overloaded cases

```
def square(x):
    """Squares x.

    >>> square(2)
    4
    >>> square(-2)
    4
    >>> square(complex(0,1))
    (-1+0j)
    """

    return x * x

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

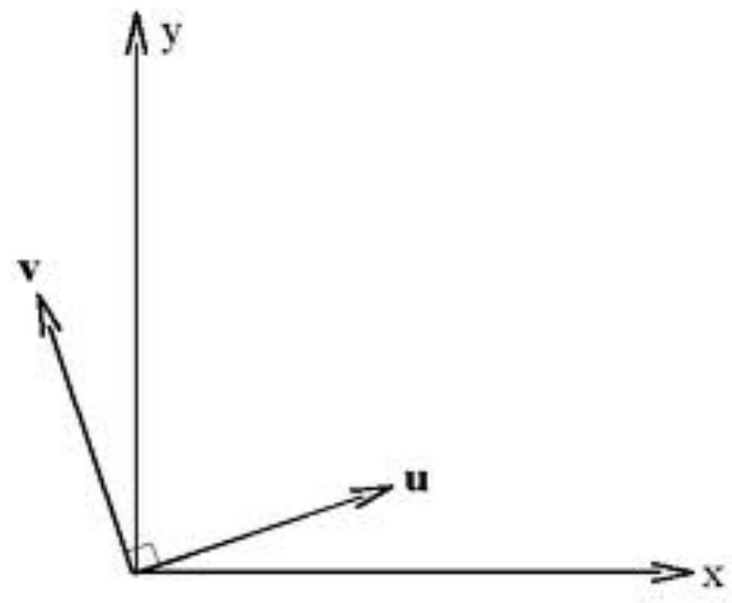
Efficiency

- Working with strength
- Proper data structures
- Avoiding weaknesses
- Dealing with version changes (backward compatibility) [python 2.X and 3.X!]

Maintainability

- More time than writing
- You don’t understand your own code
  - Comment amply
- You yourself will maintain it
- Consistent practices
  - Braces, brackets, spaces
  - Line lengths, tabs, blank lines

- (non)Duplication
- Orthogonality
- Refactoring



Duplication

- Don't repeat yourself
- Impatience
- Reinventing wheels

Don’t forget the cheat-sheets

Visit the Python cheese-shop

Also visit the Hitch Hikers Guide to Python

Orthogonality

- Decouple routines
- Make them independent
- Change in one should not affect the other
- Changes are localized
- Unit testing is easy
- Reuse is easy
- If requirements change for one function, how many modules should be affected? 1
- Configurable

```
def line(startpoint, endpoint, length):
        some code here
        …

def line2(startpoint, endpoint):
  length = endpoint – startpoint
some code here
…
```

- if while entertaining libraries you need to write/handle special code, it is not good.
- avoid global data
- avoid similar functions
- even if you are coding for a particular flavor of a particular OS, be flexible

Refactoring

- Early and often
  - Duplication
  - Non-orthogonal design
  - Outdated knowledge
  - Performance
- Don’t add functionality at the same time
- Good tests
- Short deliberate steps

Design by contract (Eiffel, Meyer ’97)

- Preconditions
- Postconditions
- Class invariants

Be strict in what you accept
Promise as little as possible
Be lazy

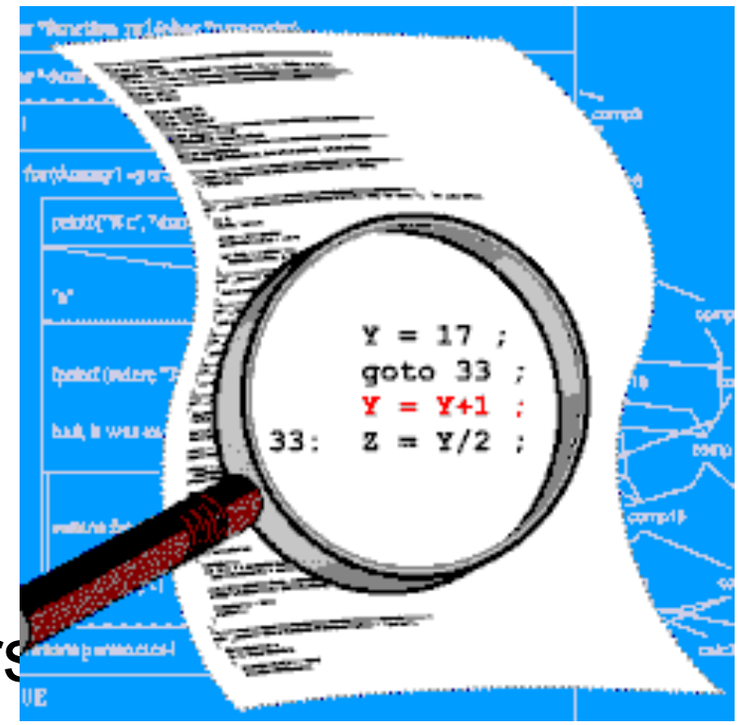
Inheritance and polymorphism result

Other aspects

- Tests
- Comments
- Arguments
- Debugging

Tests: All software will be tested
If not by you, by other users!

- Test against contract
  - Sqrt: negative, zero, string
  - Testvalue(0,0)
  - Testvalue(4,2)
  - Testvalue(-4,0)
  - Testvalue(1.e12,1000000)
- Test harness
  - Standardize logs and errors
- Test templates
- Write tests that fail



http://ib.ptb.de/8/85/851/sps/swq/graphix

things to keep in mind

- long sub names
  - test_square_of_number_2()
  - test_square_negative_number()
- standalone code
- standalone datasets
- Cleaning
  - setUp()
  - tearDown()

Python testing

- unittest – unit tests
- doctest – within your docstrings
- pytest – simpler mechanism
- nose
- tox
- mock

http://python-guide.readthedocs.org/en/latest/writing/tests/

Comments

- If it was difficult to write, it must be difficult to understand (??)
- bad code requires more comments
- tying documentation and code

Don't do this:

```
x = x + 1                 # Increment x
```

But sometimes, this is useful:

```
x = x + 1                 # Compensate for border
```

Documentation/comments in code

- List of functions exported
- Revision history
- List of other files used
- Name of the file

Documentation

- Algorithmic:

full line comments to explain the algorithm

- Elucidating: # end of line comments
- Defensive:# Has puzzled me before. Do this.
- Indicative: # This should rather be rewritten
- Discursive: # Details in POD

Arguments and return values

- Don’t let your subroutines have too many arguments
  - universe(G,e,h,c,phi,nu)
- Look for missing arguments
- Set default argument values (*args, **kwargs)

- Use explicit return values (rather than just side-effects)

notebook: arguments

Arguments

```
s = '--condition=foo --testing --output-file abc.def -x a1 a2'
args = s.split()
args
```

```
['--condition=foo', '--testing', '--output-file', 'abc.def', '-x',
```

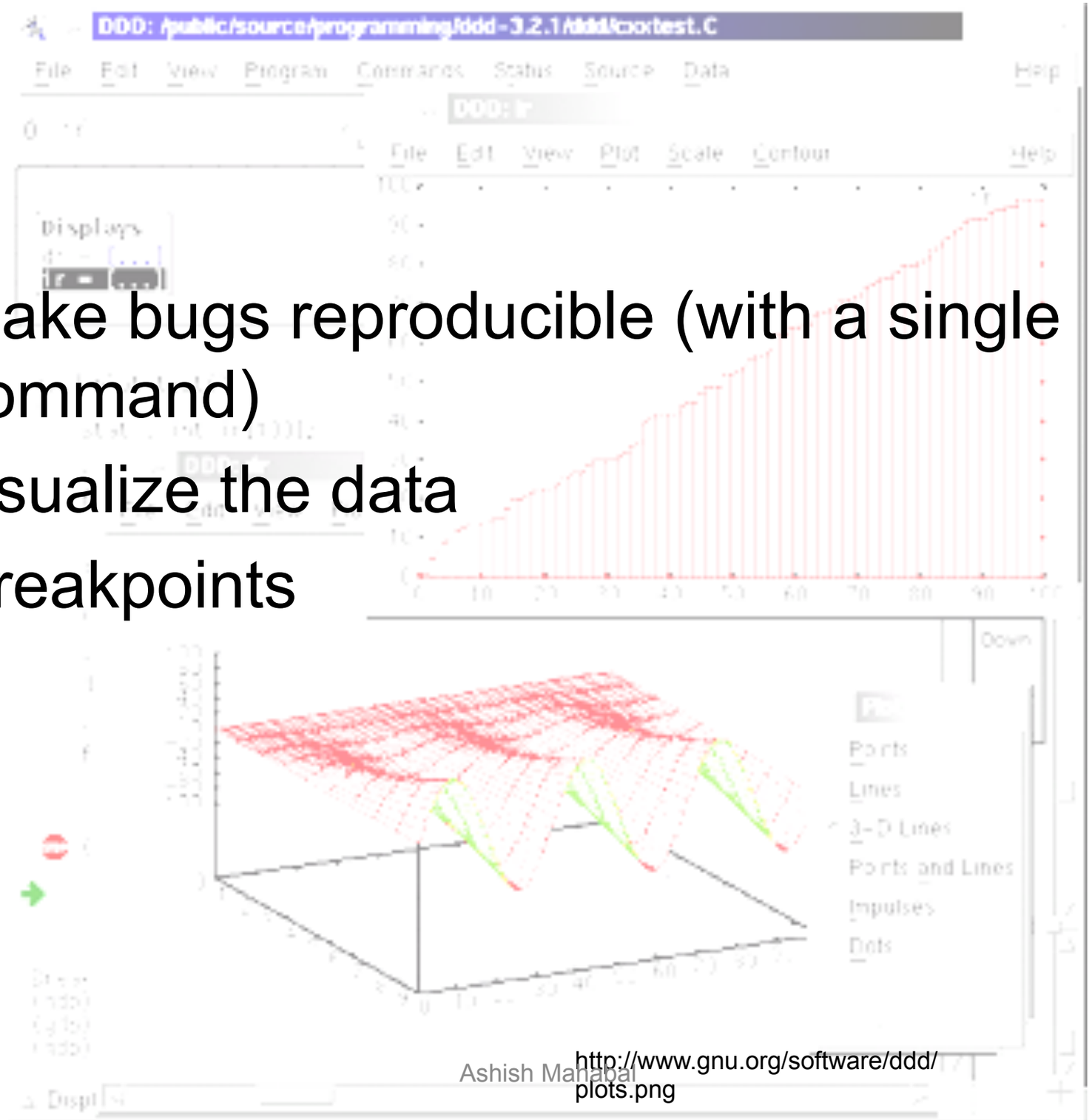
```
optlist, args = getopt.getopt(args, 'x', [
...     'condition=', 'output-file=', 'testing'])
optlist
```

```
[('--condition', 'foo'),
 ('--testing', ''),
 ('--output-file', 'abc.def'),
 ('-x', '')]
```

Debugging

- There will be bugs!
- The only bug-free program is one that does not do anything
- Tests: write unit tests first
- Make sure the program ‘compiles’ without warnings

- make bugs reproducible (with a single command)
- visualize the data
- Breakpoints



http://www.gnu.org/software/ddd/plots.png

When you find a bug …

- Check boundary conditions
  - first and last elements of lists
- Describe the problem to someone else
- Why wasn't it caught before
- Could it be lurking elsewhere (orthogonality!)
- If tests ran fine, are the tests bad?

Metaprogramming

- Configure
- Abstraction in code, details in metadata
  - Decode design
  - docstrings

```
"""Return a foobang

Optional plotz says to frobnicate the bizbaz first.

"""
```

Portfolio building

- learn general tools, invest in different ones
  - plain text
    - easier to test (config files, for instance)
  - Shells
    - find, sed, awk, grep, locate
    - .tcshrc, .Xdefaults
  - learn different (types of) languages
  - Editor
    - if you know emacs, learn just a little bit of vi (or sublime)
    - Configurable, extensible, programmable (cheat sheet)
      - syntax highlighting
      - auto completion
      - auto indentation
      - Boilerplates
      - built-in help

Text manipulation

perl and ruby are very powerful

- Code generators
  - make files, config files, shell scripts., …
- Active code generator:
  - Skyalert (transient astronomy streams)
    - new transient
    - obtain distributed archival data
    - incorporate it
    - if certain conditions met,
      - run other programs
      - or raise alerts
      - drive other telescopes
      - and obtain feedback

- Languages/tools/OSes/editors
  - 99 bottles of beer
  - Programming shootout
  - Project Euler
    - Python
    - Perl
    - J
    - Haskell

Exercise

- Write a program to count number of ways to split an amount using coins of denominations 1,5,10,25
- For numbers 1 through 100, sum those for which the answer to the first question is an odd number
- Is it odd or even?

Exercise

- Duplicate as much as possible the following using only Unix commands:

  http://lifehacker.com/5898720/a-better-strategy-for-hangman

Number of letters	Optimal calling order
1	A I
2	A O E I U M B H
3	A E O I U Y H B C K
4	A E O I U Y S B F
5	S E A O I U Y H
6	E A I O U S Y
7	E I A O U S
8	E I A O U
9	E I A O U

What are the lessons?

- Chain as weak as its weakest link
- Comment! For others and for yourself
- Tests!
- Orthogonality
- Don’t duplicate
- Designing by contract
- Know the features

Law 1: Every program can be optimized to be smaller.

Law 2: There's always one more bug.

Corollary: Every program can be reduced to a one-line bug.

Follow the Best Practices, and have fun coding