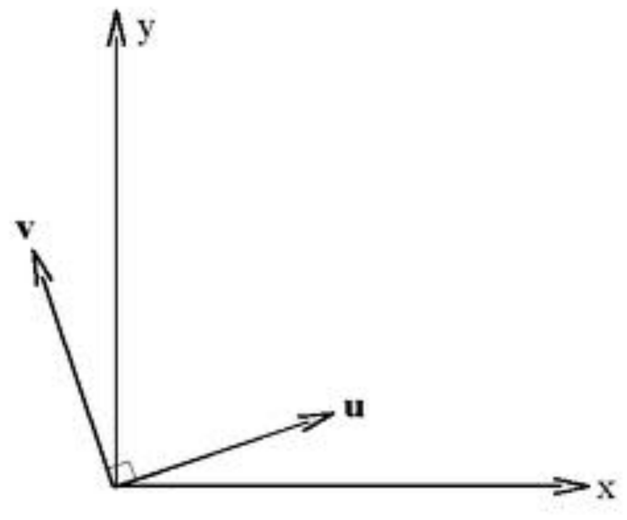
Ashish Mahabal
California Institute of Technology

Best Programming Practices - III

Sponsored by the Jet Propulsion Laboratory and by the Keck Institute for Space Studies, California Institute of Technology

- (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  - Outdated knowledge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

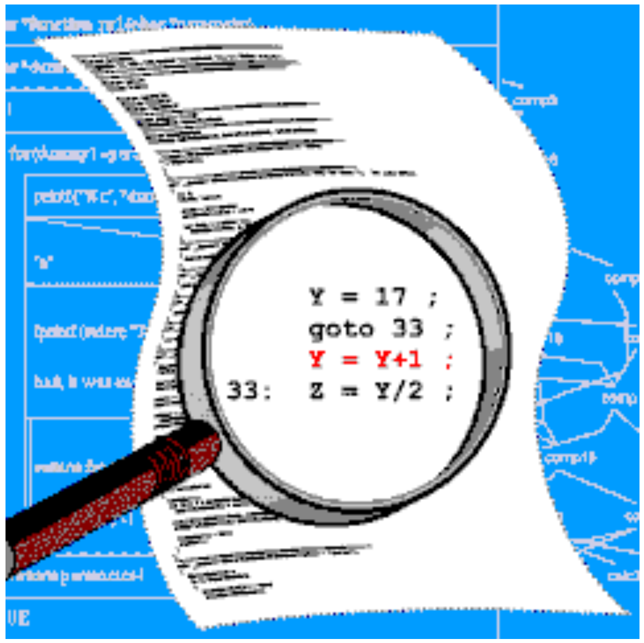
Inheritance and polymorphism result

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()

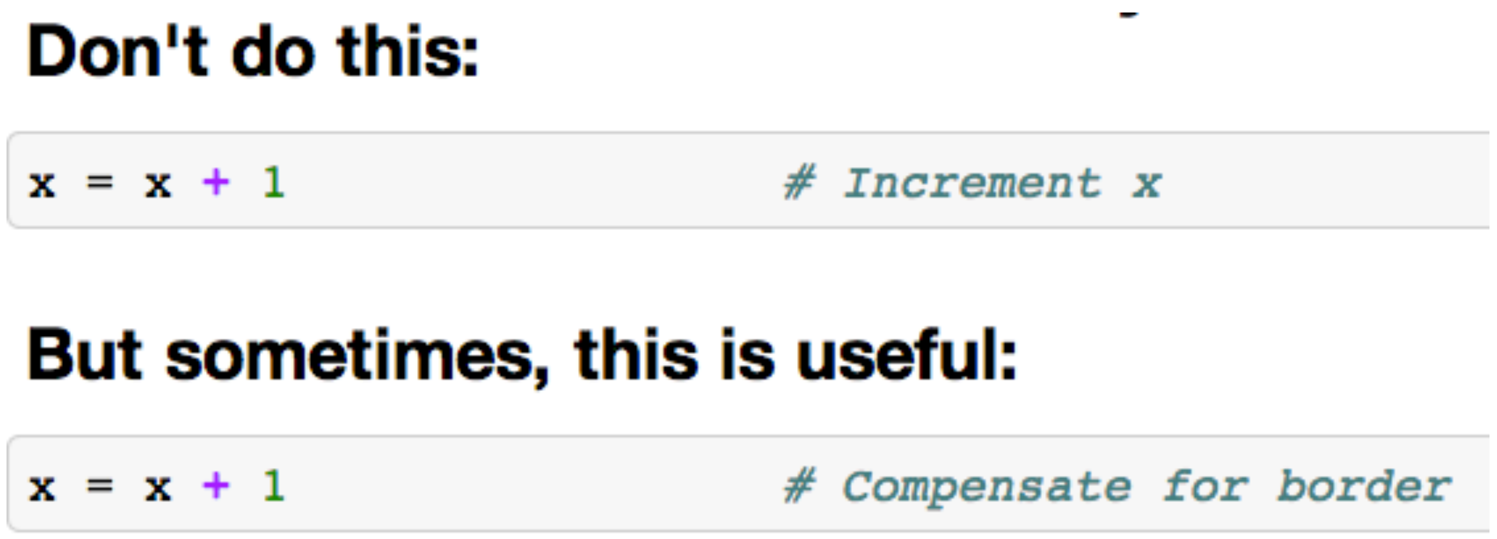
Python testing

- 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

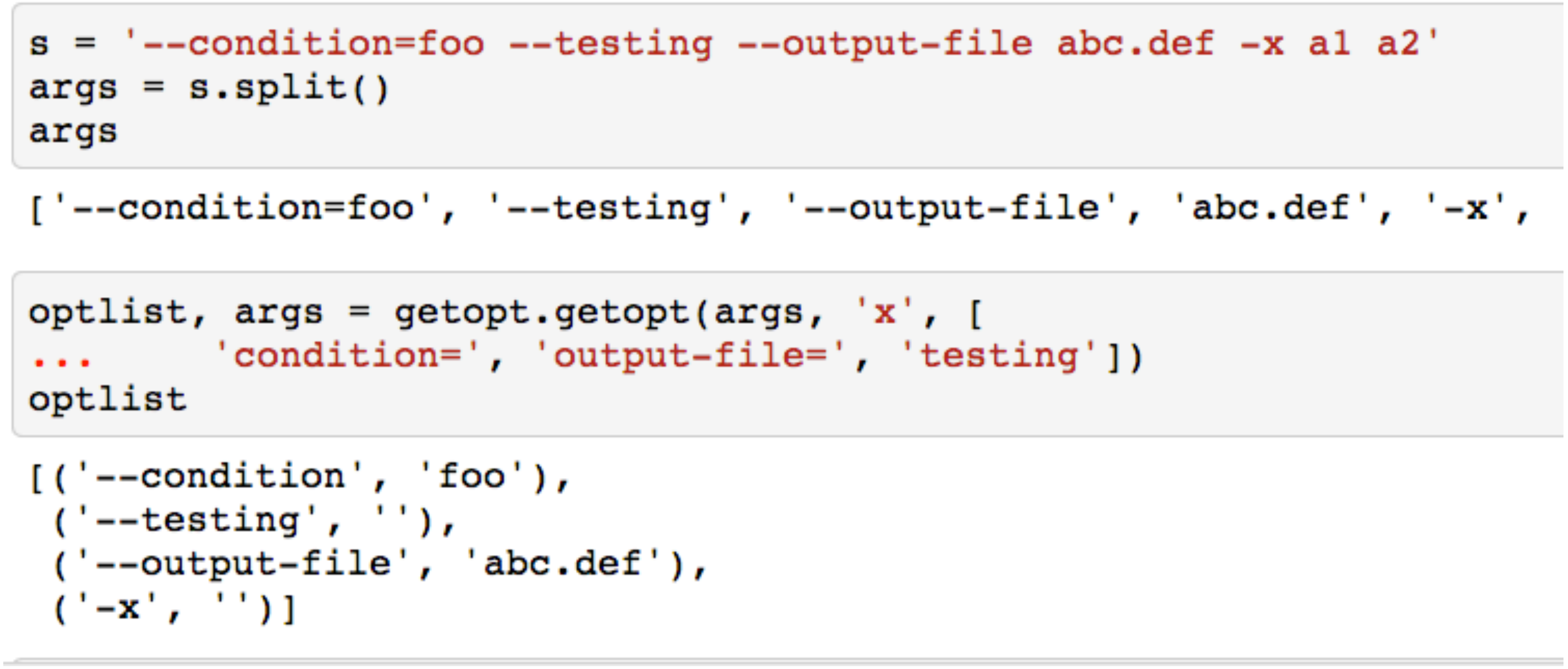
Arguments and return values

- 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
```

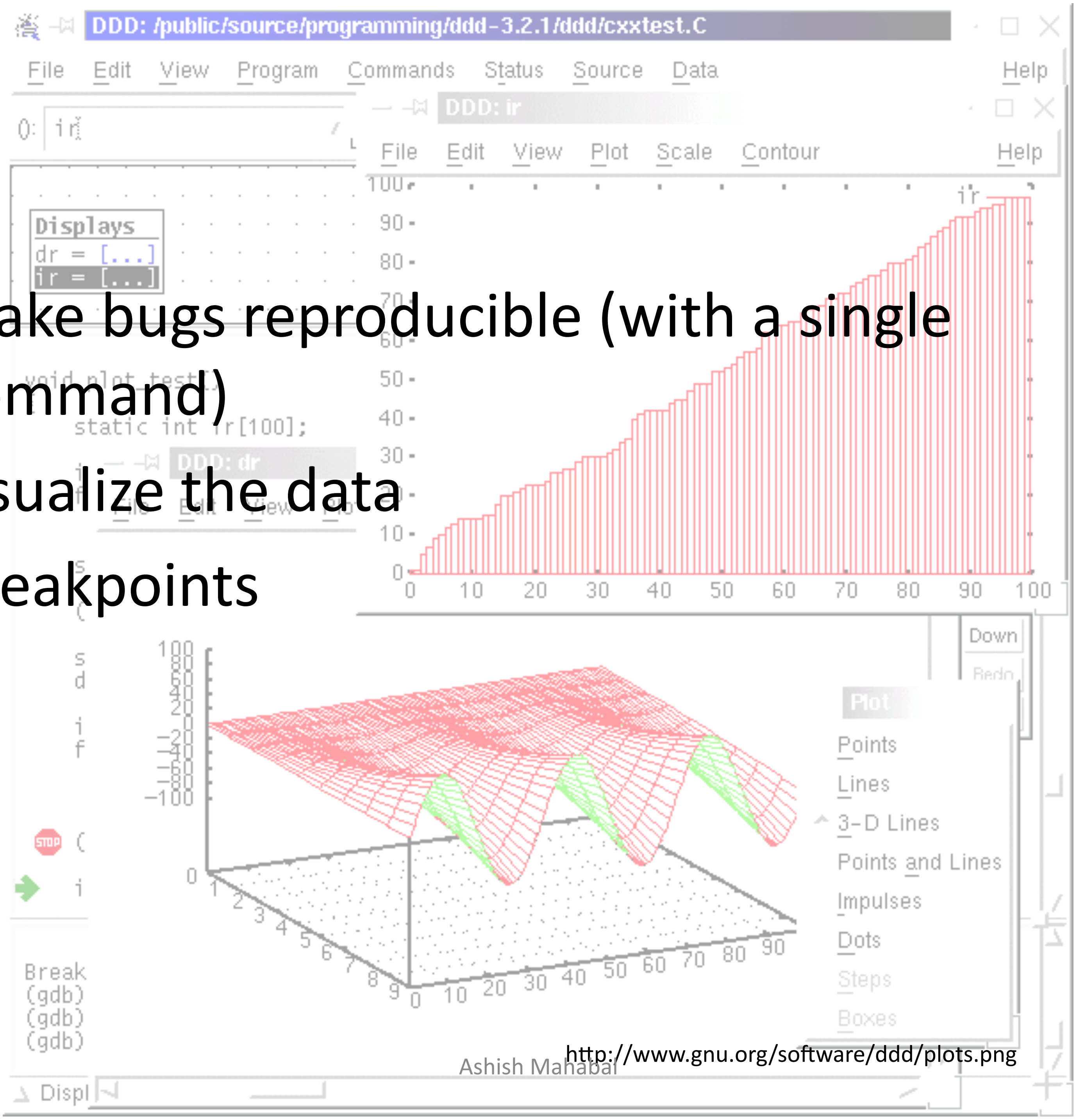
```
['--condition=foo', '--testing', '--output-file', 'abc.def', '-x',
```

```
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


DDD: /public/source/programming/ddd-3.2.1/ddd/cxxtest.C
File Edit View Program Commands Status Source Data Help
DDD: ir
File Edit View Plot Scale Contour Help
Displays
dr = [...]
ir = [...]
DDD: dr
Plot
Points
Lines
3-D Lines
Points and Lines
Impulses
Dots
Steps
Boxes
Down
Redo
Break
(gdb)
(gdb)
(gdb)

```
void plot_test()
    static int ir[100];
```

- 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?

Next time …

- Metaprogramming
- Portfolio building