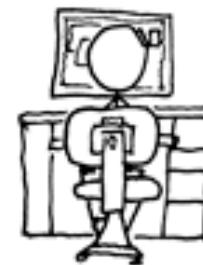


Best Practices for Scientific Coding

I COULD RESTRUCTURE
THE PROGRAM'S FLOW
OR USE ONE LITTLE
'GOTO' INSTEAD.

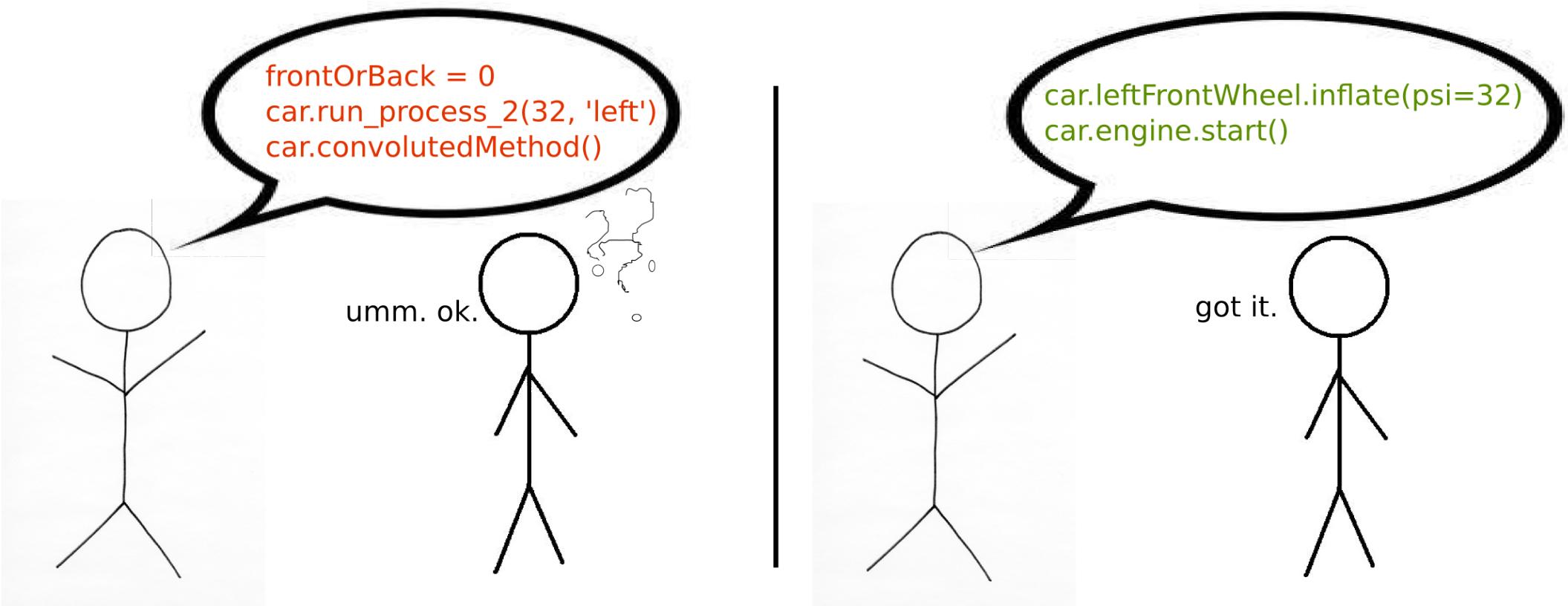


EH, SCREW GOOD PRACTICE.
HOW BAD CAN IT BE?



DATA SCIENCE BOOTCAMP

Write programs for people, not computers



The reader shouldn't have to remember tons
Only a handful of facts in their memory



The reader shouldn't have to remember tons
Only a handful of facts in their memory

Easily understood functions

Each conducts a single, easily understood task

Names should be consistent, distinctive, and meaningful



M. Night
Shyamalan



Zach
Galifianakis



Irmak
Sirer

Names should be consistent, distinctive, and meaningful

What these functions do is very clear:

- `bank_account.deposit(20, '$')`
- `bank_account.withdraw(10, '$')`

Aptly named variables:

- `current_bank_account_balance`
- `number_of_bunnies_in_cage`

Code style and formatting should be consistent

Don't use both of these in the same code:

- CamelCaseNaming
- snake_case_naming

Use the same indentation sizes

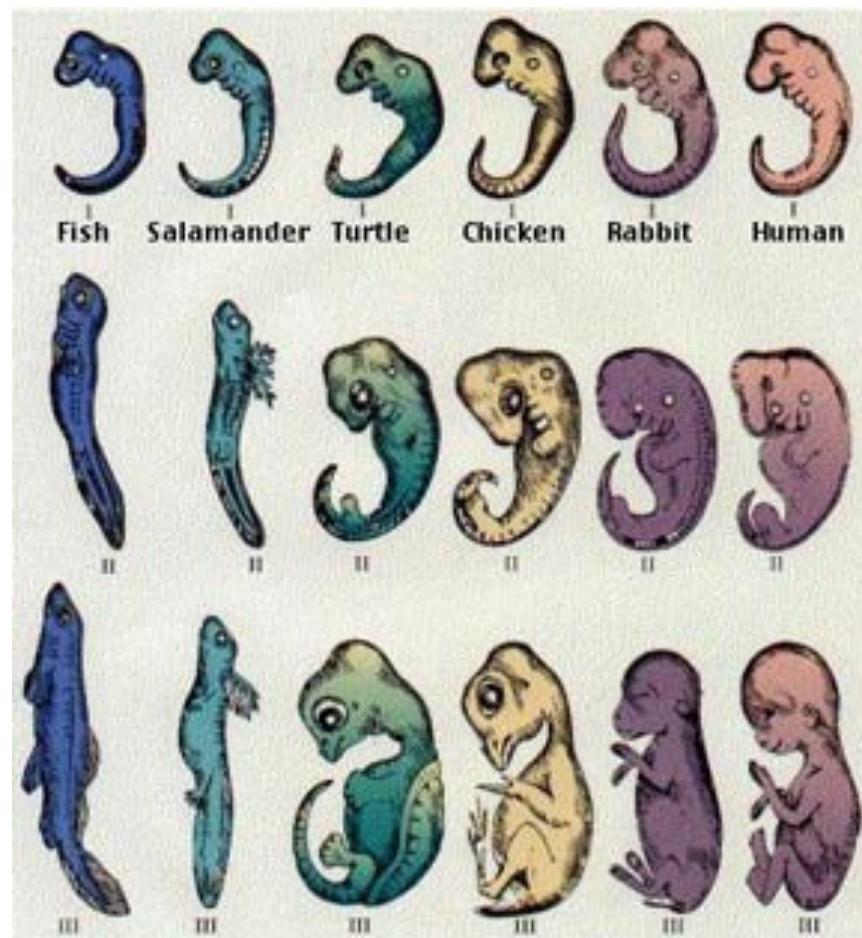
Keep max line lengths the same

All aspects of software development should be broken down into tasks roughly an hour long

Work in chunks of 50-200 lines of code

Short functions (2-15 lines)

Make incremental changes



Work in small steps
with frequent feedback and course correction

Hour long coding sessions

Grouped in iterations

Produce working but incomplete code (or results)
after each iteration

Use version control



Use a version control system



git

Everything that's been created manually
should be put in version control



Everything that's been created manually
should be put in version control

Not data

Everything that's been created manually
should be put in version control

Not data

Not tons of big image or text files that don't change

Everything that's been created manually should be put in version control

Not data

Not tons of big image or text files that don't change

Not automatically created stuff like `.pyc` files or
emacs backups like `script.py~`

Everything that's been created manually should be put in version control

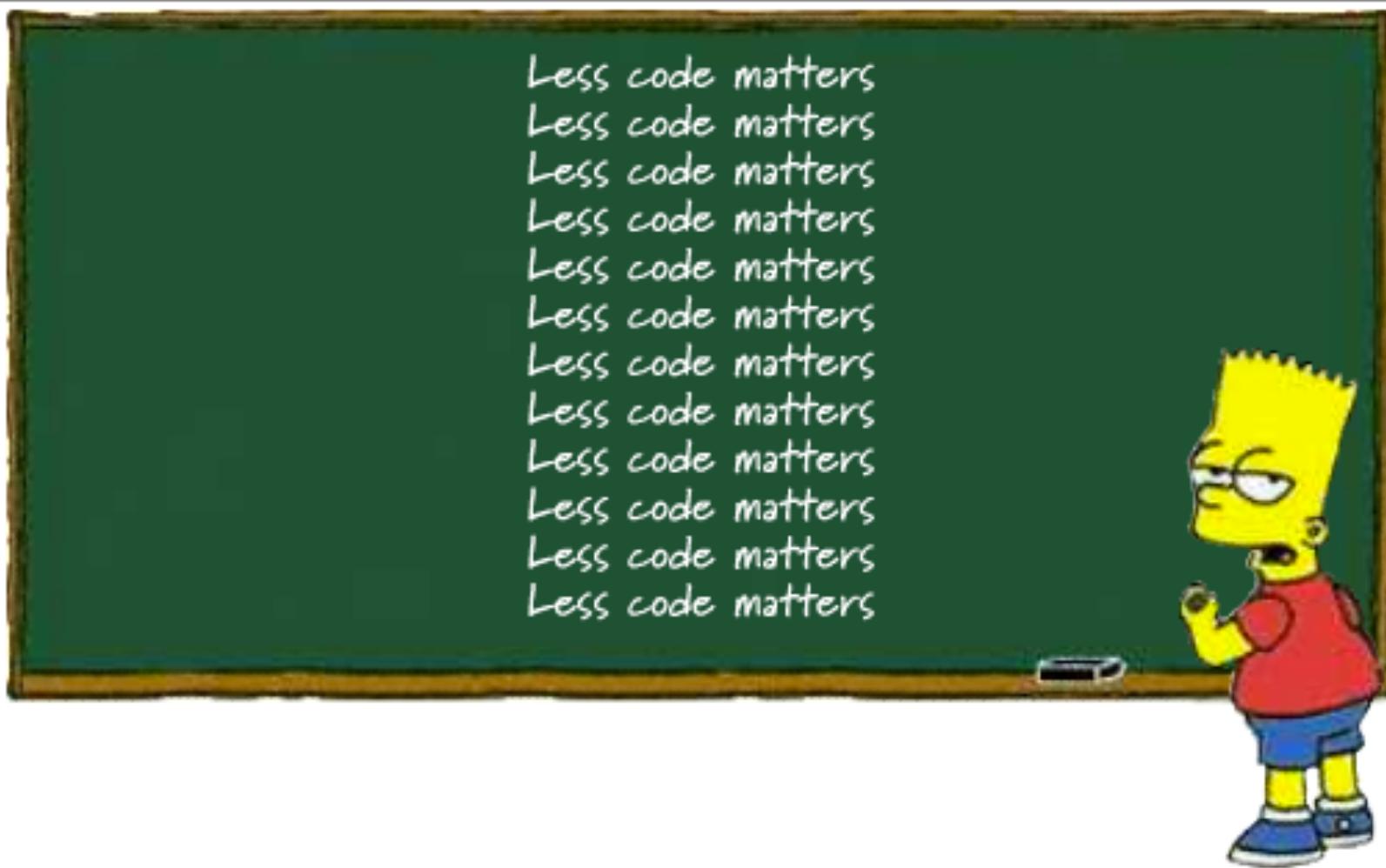
Not data

Not tons of big image or text files that don't change

Not automatically created stuff like `.pyc` files or
emacs backups like `script.py~`

Commit result graphs

Don't repeat yourself (or others)



Small scales: don't copy & paste, modularize
DO NOT copy-paste. I mean it.

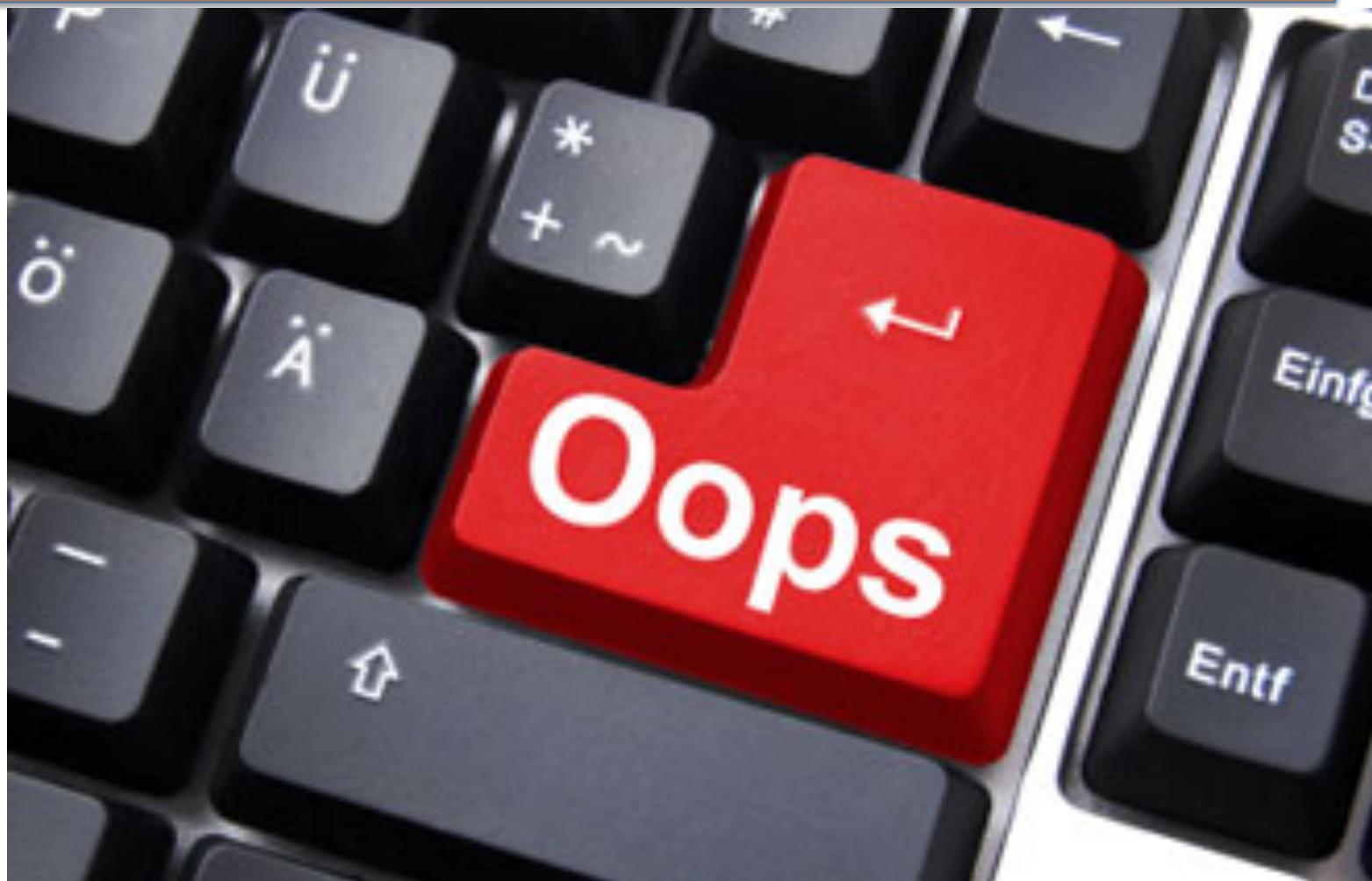


Large scales: Look for existing packages

Don't reinvent the wheel

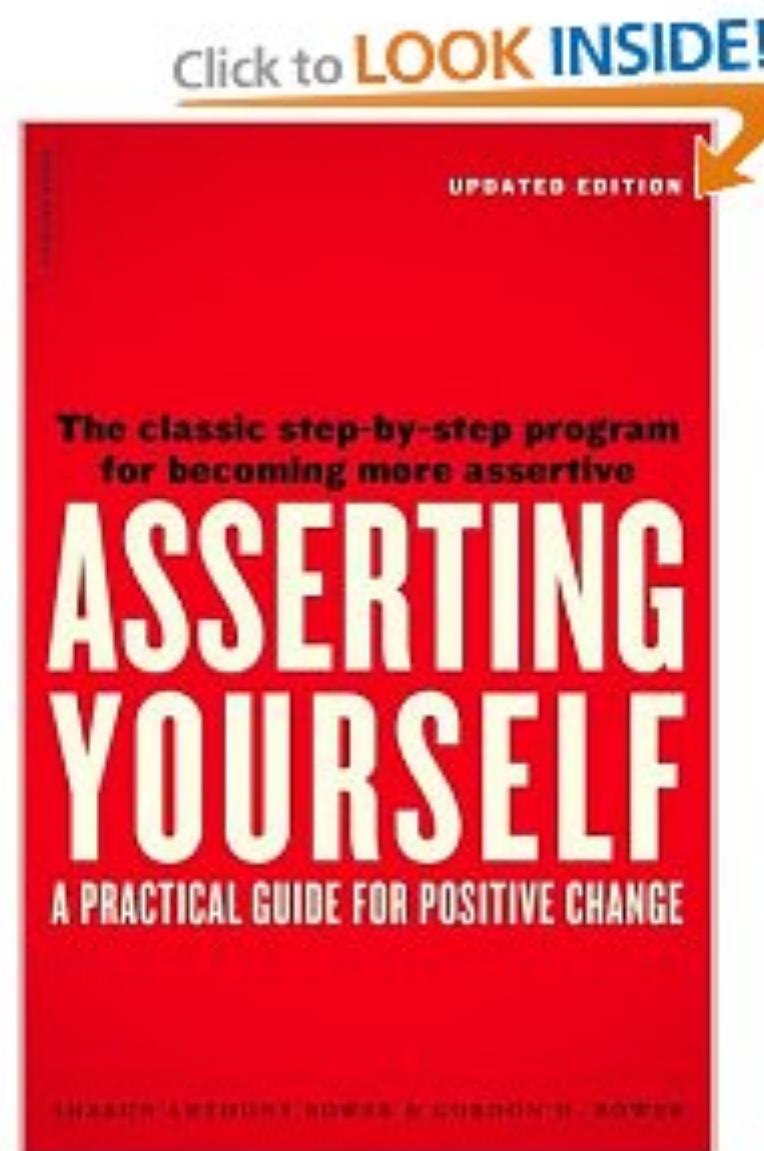


Plan for mistakes



Defensive programming

Add assertions to programs to check assumptions



Defensive programming

Add assertions to programs to check assumptions

```
def bradford_transfer(grid, point, smoothing):
```

```
    assert grid.contains(point), 'Point is not located in grid'
```

```
    assert grid.is_local_maximum(point), 'Point is not a local max in grid'
```

```
    ...do calculations...
```

```
    assert 0.0 < result <= 1.0, 'Bradford transfer value out of legal range'
```

```
    return result
```

Defensive programming

Add assertions to programs to check assumptions

```
prev->next = toDelete->next;  
delete toDelete;
```

```
// if only forgetting were  
// this easy for me.
```



```
assert "It's going to be okay.";
```



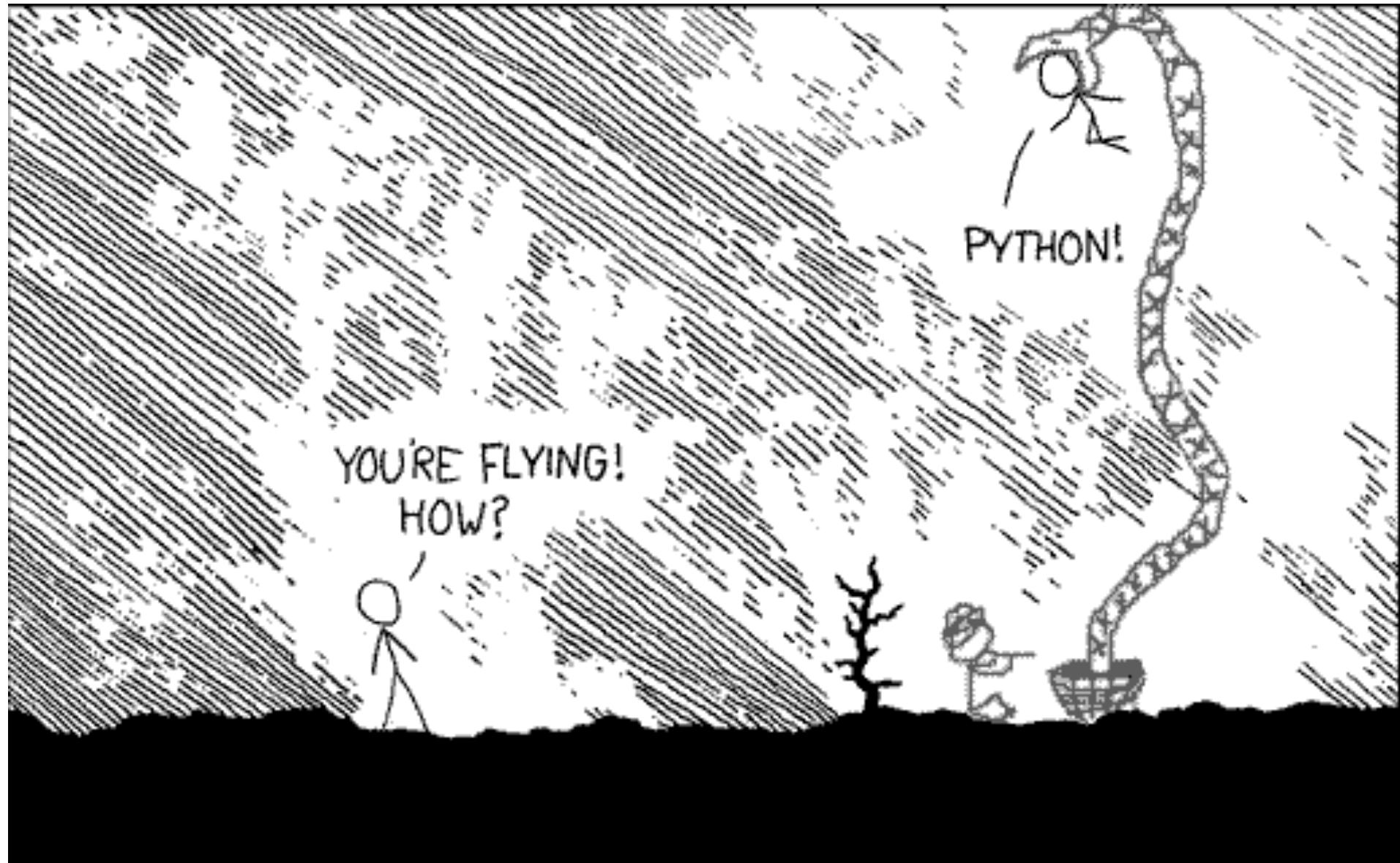
Turn bugs into test cases



Optimize software only after it works correctly

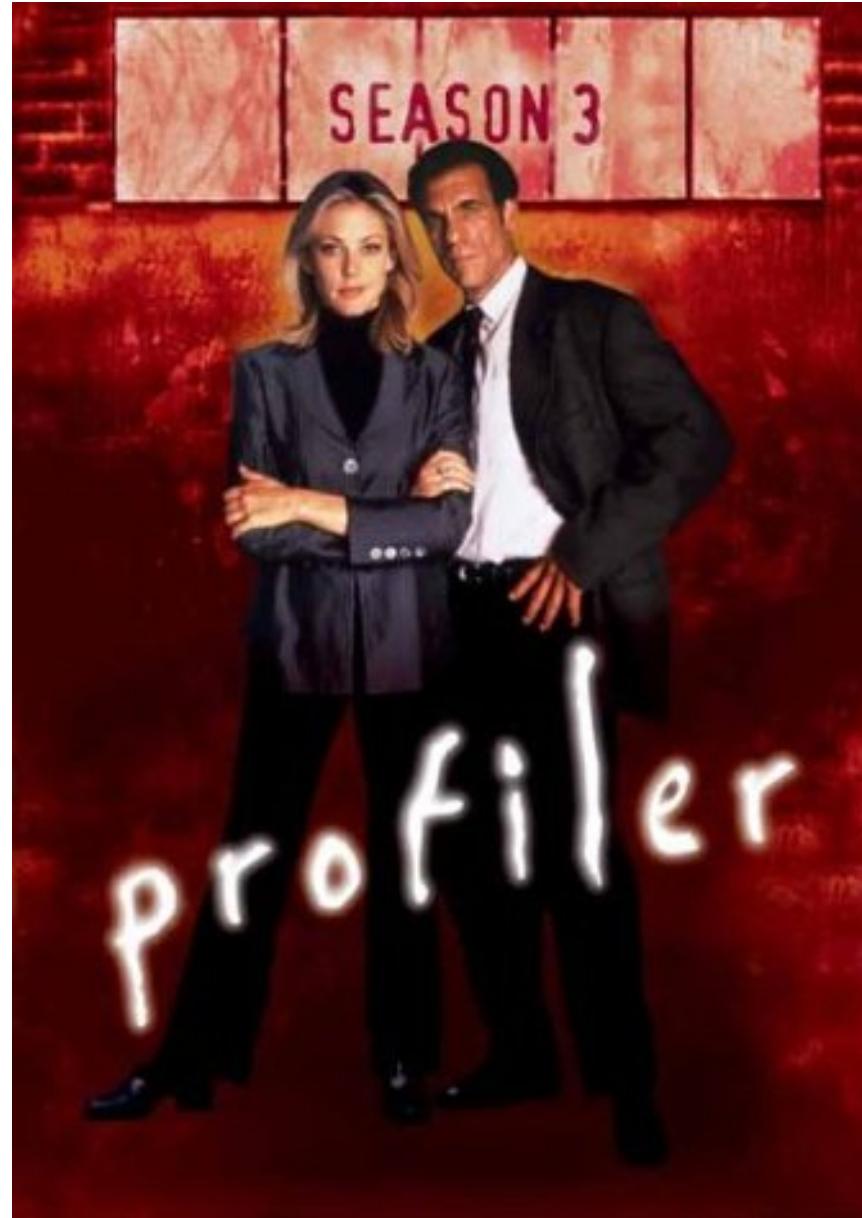


Develop in the highest-level language possible

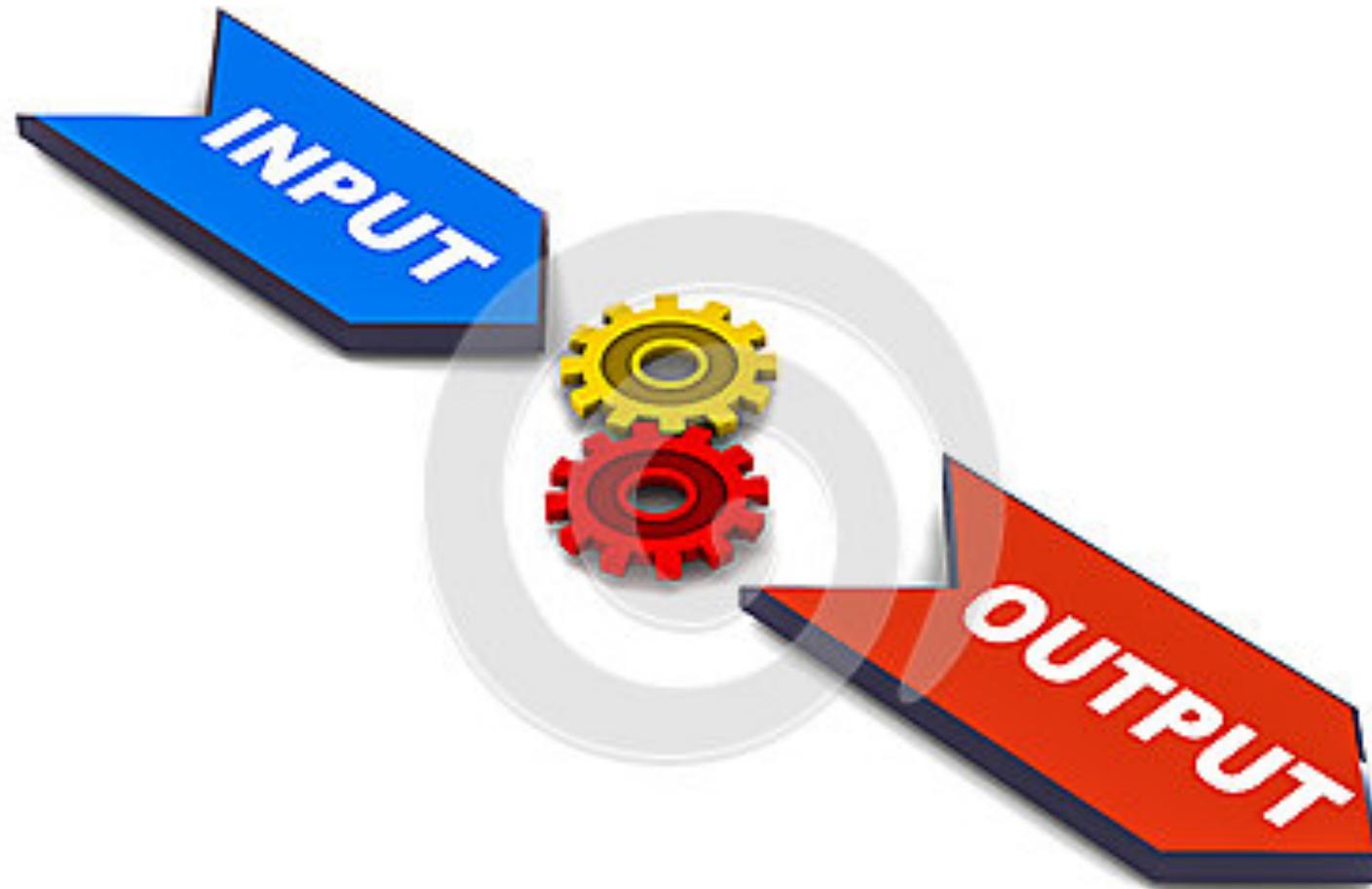


Use a profiler to identify bottlenecks

A profiler times every step of a piece of code

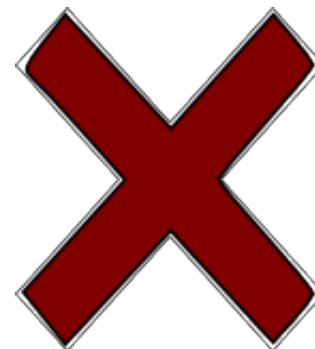


Document interfaces and reasons not implementations



Document interfaces and reasons not implementations

i = i + 1 # Increment the variable i by one.



Include documentation for code within the code

Comment often

```
1
2 class Point:
• 3     """ Point class represents and manipulates x,y coords. """
4
5     def __init__(self, x=0, y=0):
• 6         """ Create a new point at x, y """
• 7         self.x = x
• 8         self.y = y
9
10 # Other statements outside the class continue below here.
11
• 12 q = Point(
    x=0, y=0
    Create a new point at x, y
```

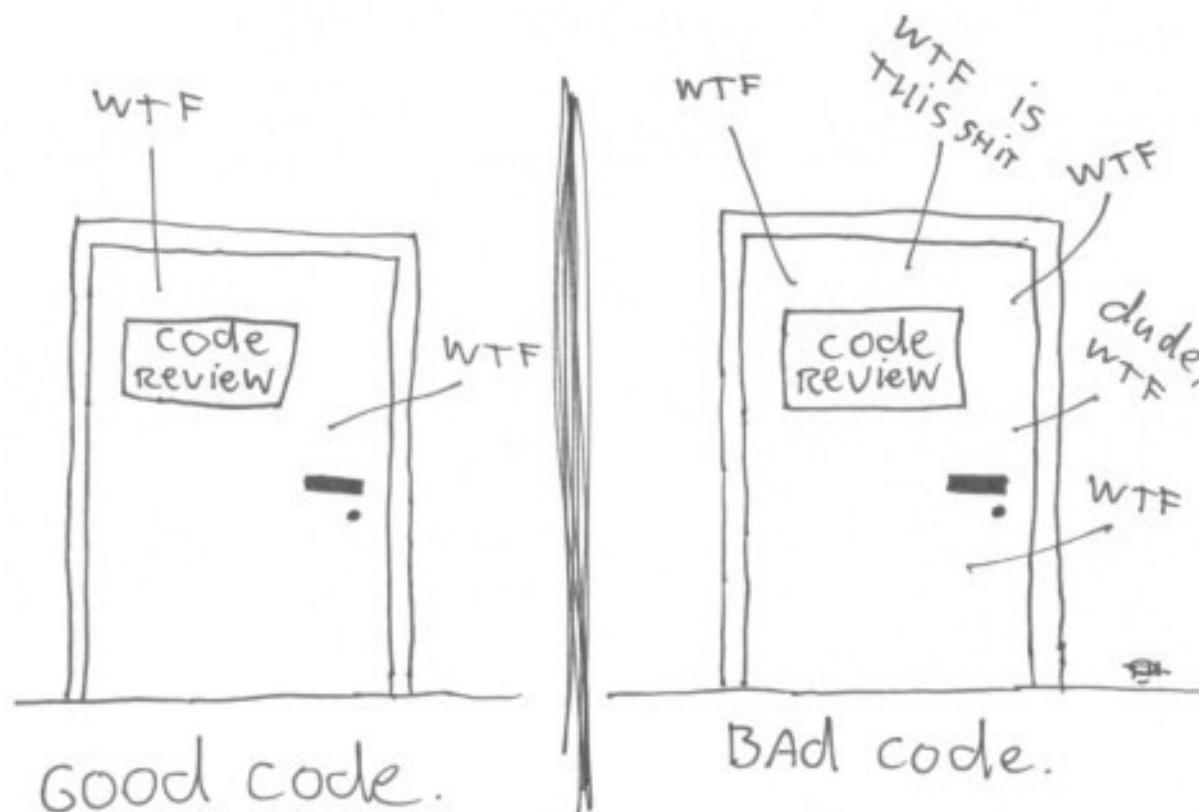
Include documentation for code within the code
Comment often

Docstrings: How to use code

Comments: Why & how code works

Conduct code reviews

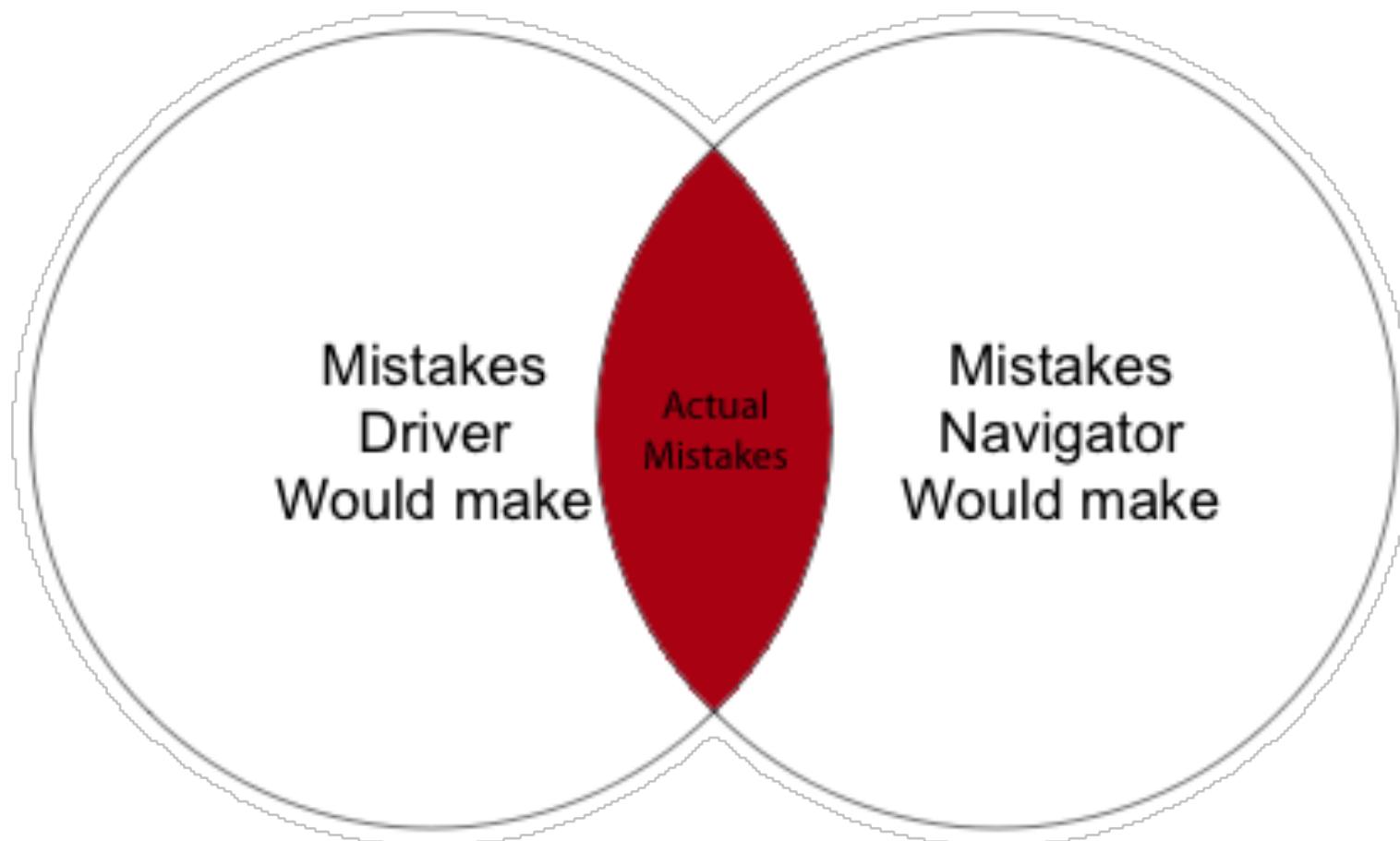
The ONLY VALID MEASUREMENT
OF CODE QUALITY: WTFs/MINUTE



Pair Programming



Pair Programming



Pair Programming

Tackling tricky design/coding/debugging

Bringing someone new up to speed

Whenever you can

Python Conventions

Moment of Zen

Simple is better
than complex



pluralsight

Explicit Code No Black Magic

Bad

```
def make_complex(*args):
    x, y = args
    return dict(**locals())
```

Explicit Code No Black Magic

Bad

```
def make_complex(*args):
    x, y = args
    return dict(**locals())
```

Good

```
def make_complex(x, y):
    return {'x': x, 'y': y}
```

One Statement Per Line

One idea at a time

Bad

```
print 'one'; print 'two'
```

```
if x == 1: print 'one'
```

```
if <complex comparison> and <other complex comparison>:  
    # do something
```

One Statement Per Line

One idea at a time

Good

```
print 'one'  
print 'two'  
  
if x == 1:  
    print 'one'  
  
cond1 = <complex comparison>  
cond2 = <other complex comparison>  
if cond1 and cond2:  
    # do something
```

Naming conventions

What to call variables

snake_case for functions, methods, attributes

snake_case or SCREAM_CASE for constants

CapsCamelCase for classes

Naming conventions

What to call variables

snake_case for functions, methods, attributes

snake_case or SCREAM_CASE for constants

CapsCamelCase for classes

regular_attribute

_internal_attribute

__private_attribute

Whitespace Formatting neat code

4 spaces per indentation level

Never mix tabs and spaces

One blank line between functions

Two blank lines between classes

Long Lines

Formatting neat code

Keep lines shorter than 80 characters

Use parentheses to spread over multiple lines

Long Lines

Formatting neat code

Keep lines shorter than 80 characters

Use parentheses to spread over multiple lines

Bad:

```
my_very_big_string = """For a long time I used to go to bed early. Sometimes, \
    when I had put out my candle, my eyes would close so quickly that I had not even \
        time to say "I'm going to sleep.""""

from some.deep.module.inside.a.module import a_nice_function, another_nice_function, \
    yet_another_nice_function
```

Long Lines

Formatting neat code

Keep lines shorter than 80 characters

Use parentheses to spread over multiple lines

Good:

```
my_very_big_string = (
    "For a long time I used to go to bed early. Sometimes, "
    "when I had put out my candle, my eyes would close so quickly "
    "that I had not even time to say \"I'm going to sleep.\""
)

from some.deep.module.inside.a.module import (
    a_nice_function, another_nice_function, yet_another_nice_function)
```

Python Idioms



Check if variable equals a constant

No need to explicitly compare to True, None, 0

Bad:

```
if attr == True:  
    print 'True!'  
  
if attr == None:  
    print 'attr is None!'
```

Check if variable equals a constant

No need to explicitly compare to True, None, 0

Good:

Just check the value

```
if attr:  
    print 'attr is truthy!'
```

or check for the opposite

```
if not attr:  
    print 'attr is falsey!'
```

or, since None is considered false, explicitly check for it

```
if attr is None:  
    print 'attr is None!'
```

Access a Dictionary Element

Use `x in d` instead of `has_key`

Bad:

```
d = {'hello': 'world'}
if d.has_key('hello'):
    print d['hello']      # prints 'worlD'
else:
    print 'default_value'
```

Access a Dictionary Element

Use `x in d` instead of `has_key`

Good:

```
d = {'hello': 'world'}
```

```
print d.get('hello', 'default_value') # prints 'world'  
print d.get('thingy', 'default_value') # prints 'default_value'
```

Or:

```
if 'hello' in d:  
    print d['hello']
```

Manipulating lists

List comprehensions or map() and filter()

Bad:

```
# Filter elements greater than 4
a = [3, 4, 5]
b = []
for i in a:
    if i > 4:
        b.append(i)
```

Manipulating lists

List comprehensions or map() and filter()

Good:

```
a = [3, 4, 5]
b = [i for i in a if i > 4]
# Or:
b = filter(lambda x: x > 4, a)
```

Manipulating lists

List comprehensions or map() and filter()

Bad:

```
# Add three to all list members.  
a = [3, 4, 5]  
for i in range(len(a)):  
    a[i] += 3
```

Manipulating lists

List comprehensions or map() and filter()

Good:

```
a = [3, 4, 5]
a = [i + 3 for i in a]
# Or:
a = map(lambda i: i + 3, a)
```

Read from a file

Use the with statement

Bad:

```
f = open('file.txt')
a = f.read()
print a
f.close()
```

Read from a file

Use the with statement

Good:

```
with open('file.txt') as f:  
    for line in f:  
        print line
```