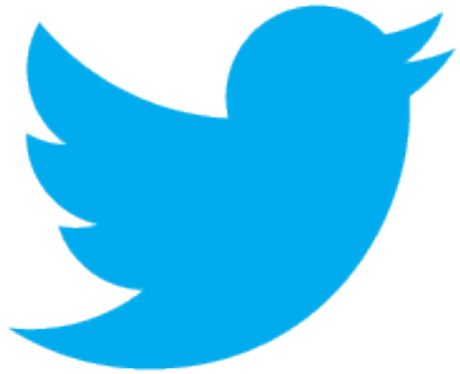
Thinking Hard About Python

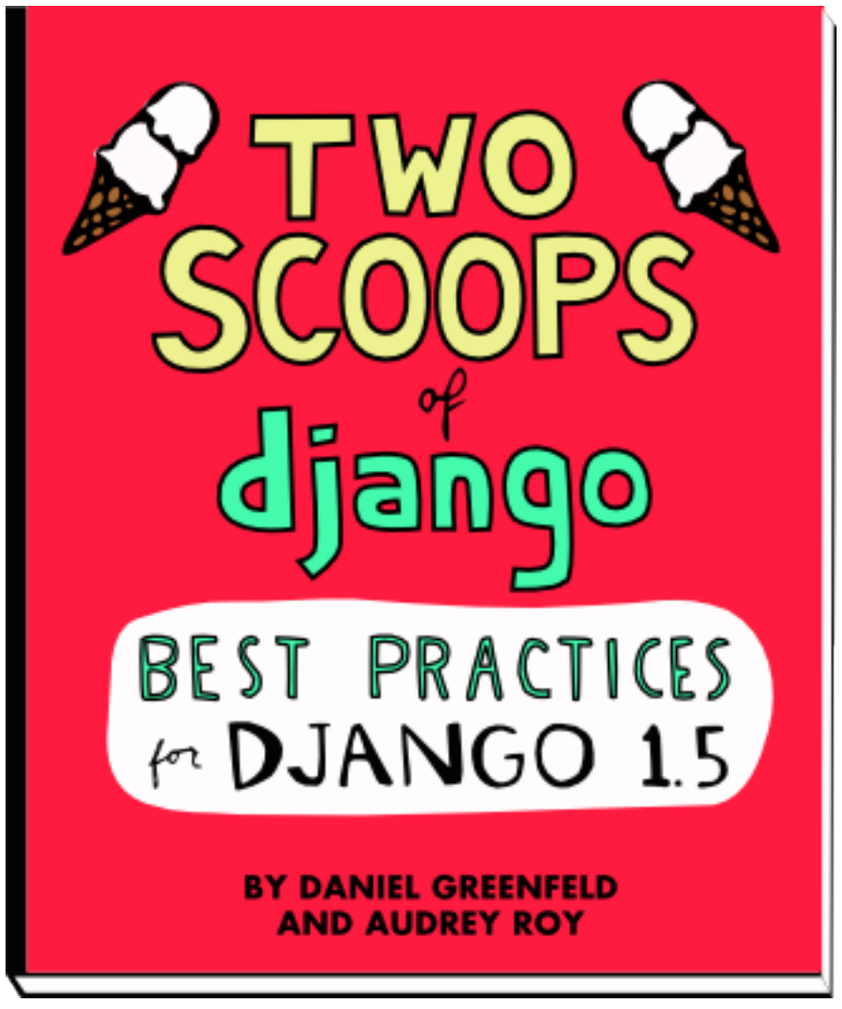
Daniel Greenfeld

Daniel Greenfeld
pydanny.com / @pydanny

@pydanny



Daniel Greenfeld
pydanny.com / @pydanny



Danny: 128,546++

Audrey: 121,871++

http://2scoops.org

Daniel Greenfeld
pydanny.com / @pydanny

What I want you to think of me.

One Cartwheel of Many Around the World

Daniel Greenfeld
pydanny.com / @pydanny

What I'm really like.

Myself at 13 in front of the Apple][

Daniel Greenfeld
pydanny.com / @pydanny

Overview

Daniel Greenfeld
pydanny.com / @pydanny

Coding into Trouble

Daniel Greenfeld
pydanny.com / @pydanny

Controversy


Daniel Greenfeld
pydanny.com / @pydanny

Exceptions


Daniel Greenfeld
pydanny.com / @pydanny

Avoiding Technical Debt


Daniel Greenfeld
pydanny.com / @pydanny

...first section...

Daniel Greenfeld
pydanny.com / @pydanny

Coding into Trouble

a.k.a.

Daniel Greenfeld
pydanny.com / @pydanny

Super()

Troubles

Circle

```
import math
class Circle(object):

    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return self.radius ** 2 *math.pi

    def __repr__(self):
        return '{0} as area {1}'.format(
            self.__class__.__name__, self.area()
        )

class Donut(Circle):

    def __init__(self, outer, inner):
        super().__init__(outer)
        self.inner = inner

    def area(self):
        outer, inner = self.radius, self.inner
        return Circle(outer).area() - Circle(inner).area()
```

```
>>> Circle(10)
Circle as area
314.159265359

>>> Donut(10, 5)
235.619449019
```

The super method calls the parent class, which is Circle

Superclassing is so easy!

What if our inheritance isn't simple?

Daniel Greenfeld
pydanny.com / @pydanny

Contention

The super() method can create ambiguity.

Daniel Greenfeld
pydanny.com / @pydanny

Example:

Django

Daniel Greenfeld
pydanny.com / @pydanny

Class Based Generic Views

- Composition
- Inheritance
- Subclassing
- Polymorphism
- Lots of other big words used to impress other developers, students, your boss, your doctor, Capoiera mestre, dog, cat, friends, family, and other people who generally don't care about such things.


Daniel Greenfeld
pydanny.com / @pydanny

However...

Things I don't know:

The ancestor chain for

django.views.generic.edit.UpdateView


Daniel Greenfeld
pydanny.com / @pydanny

The ancestor chain for django.views.generic.edit.UpdateView:

```
django.views.generic.edit.UpdateView
django.views.generic.detail.SingleObjectTemplateResponseMixin
django.views.generic.base.TemplateResponseMixin
django.views.generic.edit.BaseUpdateView
django.views.generic.edit.ModelFormMixin
django.views.generic.edit.FormMixin
django.views.generic.detail.SingleObjectMixin
django.views.generic.edit.ProcessFormView
django.views.generic.base.View
```

A form_valid() implementation

```
def form_valid(self, form):
    verb_form = verb_form_base(self.request.POST)
    if verb_form.is_valid():
        form.instance.verb_attributes = verb_form.cleaned_data
    return super().form_valid(form)
```

OMG Which form_valid() am I calling?


Daniel Greenfeld
pydanny.com / @pydanny

A form_valid() implementation

OMG!

OMG!

```
class ActionUpdateView(
        LoginRequiredMixin, # django-braces
        ActionBaseView, # inherits from AuthorizedForProtocolMixin
        AuthorizedforProtocolEditMixin, # Checks rights on edit views
        VerbBaseView, # Gets one of 200+ verb forms
            UpdateView): # django.views.generic.BaseView

    def form_valid(self, form):
        verb_form = verb_form_base(self.request.POST)
        if verb_form.is_valid():
            form.instance.verb_attributes = verb_form.cleaned_data
        return super().form_valid(form)
```

OMG!

Daniel Greenfeld
pydanny.com / @pydanny

Ancestor Chain (MRO) of ActionUpdateView

```
from actions.views import ActionUpdateView
for x in ActionUpdateView.mro():
    print(x)
```

Print the MRO

MRO = Method Resolution Order

Daniel Greenfeld
pydanny.com / @pydanny

Ancestor Chain (MRO)

```
<class 'actions.views.ActionUpdateView'>
<class 'braces.views.LoginRequiredMixin'>
<class 'actions.views.ActionBaseView'>
<class 'core.views.AuthorizedForProtocolMixin'>
<class 'core.views.AuthorizedforProtocolEditMixin'>
<class 'verbs.views.VerbBaseView'>
<class 'django.views.generic.edit.UpdateView'>
<class 'django.views.generic.detail.SingleObjectTemplateResponseMixin'>
<class 'django.views.generic.base.TemplateResponseMixin'>
<class 'django.views.generic.edit.BaseUpdateView'>
<class 'django.views.generic.edit.ModelFormMixin'>
<class 'django.views.generic.edit.FormMixin'>
<class 'django.views.generic.detail.SingleObjectMixin'>
<class 'django.views.generic.edit.ProcessFormView'>
<class 'django.views.generic.base.View'>
<type 'object'>
```


Ancestor Chain (MRO) of ActionUpdateView

```
from actions.views import ActionUpdateView
for x in [x for x in ActionUpdateView.mro() if hasattr(x, "form_valid")]:
    print(x)
```

Filter the MRO list to only include classes with a form_valid() nethod

Ancestor Chain (MRO) of

Current class

super's chosen form_valid() ancestor

```
<class 'actions.views.ActionUpdateView'>
<class 'django.views.generic.edit.UpdateView'>
<class 'django.views.generic.edit.BaseUpdateView'>
<class 'django.views.generic.edit.ModelFormMixin'>
<class 'django.views.generic.edit.FormMixin'>
```

Daniel Greenfeld
pydanny.com / @pydanny

Whew!

Daniel Greenfeld
pydanny.com / @pydanny

Safe!


Daniel Greenfeld
pydanny.com / @pydanny

If you're not careful, super can cause subtle inheritance/MRO problems.


Daniel Greenfeld
pydanny.com / @pydanny

Possible mitigations for this view.

- Hope that anyone else maintaining this project isn't going to **kill** me.
- Convert to a functional view.
- Explore better patterns.
- **return** UpdateView.form_valid(self, form)

TODO

Write a easy-to-use MRO inspector ***thingee*** that identifies the parent attributes/methods specified by the coder.

Daniel Greenfeld
pydanny.com / @pydanny

Controversy

Special cases aren't special enough to break the rules.

Although practicality beats purity.*

* Zen of Python, lines 8 and 9

Daniel Greenfeld
pydanny.com / @pydanny

Zen of Python

```
$ python -c “import this”
```

The Zen of Python, by Tim Peters

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

PEP-0020


Daniel Greenfeld
pydanny.com / @pydanny

Special cases aren't special enough to break the rules.

Although practicality beats purity.*

* Zen of Python, lines 8 and 9

Daniel Greenfeld
pydanny.com / @pydanny

Web2py

Often honors Implicit over Explicit

Follows its own namespace pattern


Daniel Greenfeld
pydanny.com / @pydanny

Web2py code sample

```
# encoding: utf-8
# https://github.com/mdipierro/evote/blob/master/models/menu.py
# this file is released under public domain and
# you can use without limitations

response.title = 'Voting Service'
response.subtitle = None

## read more at http://dev.w3.org/html5/markup/meta.name.html
response.meta.author = 'Your Name <you@example.com>'
response.meta.description = 'a cool new app'
response.meta.keywords = 'web2py, python, framework'
response.meta.generator = 'Web2py Web Framework'

# snip more content that I cut in the name of brevity
```

Daniel Greenfeld
pydanny.com / @pydanny

Web2py code sample

```
# encoding: utf-8
# https://github.com/mdipierro/evote/blob/master/models/menu.py
# this file is released
# you can use without lim

response.title = 'Voting
response.subtitle = None

## read more a                                    .name.html
response.meta.
response.meta.
response.meta.
response.meta.

# snip more co                                   ty
```

I GET IT NOW

Europe taught me why unicode is important.

Daniel Greenfeld
pydanny.com / @pydanny

Web2py code sample

```
# encoding: utf-8
# https://github.com/mdipierro/evote/blob/master/models/menu.py
# this file is released under public domain and
# you can use without limitations

response.title =
response.subtitl

## read more at http://dev.w3.org/html5/markup/meta.name.html
response.meta.author = 'Your Name <you@example.com>'
response.meta.description = 'a cool new app'
response.meta.keywords = 'web2py, python, framework'
response.meta.generator = 'Web2py Web Framework'

# snip more content that I cut in the name of brevity
```

OK, Back to the talk...

Web2py code sample

```
# encoding: utf-8
# https://github.com/mdipierro/ev                    by
# this file is released under pub
# you can use without limit
response.title = 'Voting Service'
response.subtitle = None

## read more at http://dev.w3.org/ht
response.meta.author = 'Your Name <you@example.com>'
response.meta.description = 'a cool new app'
response.meta.keywords = 'web2py, python, framework'
response.meta.generator = 'Web2py Web Framework'

# snip more content that I cut in the name of brevity
```

Response object magically exists.
No import necessary

Written by Massimo himself

What about namespace pollution?

What can I expect in any location?

Daniel Greenfeld
pydanny.com / @pydanny

Contention

Web2py violates these 3 koans:

- Explicit is better than implicit
- In the name of ambiguity, refuse the temptation to guess
- Namespaces are one honking great idea -- let's do more of those!

* Zen of Python, lines 2, 12, 19

Controversy

Special cases aren't special enough to break the rules.
Although practicality beats purity.*

* Zen of Python, lines 8, 9

Web2py contends:

Special cases aren't special enough to break the rules.

Although practicality beats purity.*

* Zen of Python, lines 8, 9

Daniel Greenfeld
pydanny.com / @pydanny

Web2py contends:

Note: This is my interpretation of Web2py design considerations.

- Implicit behaviors means Web2py is easier for beginners to learn.
- The Web2py namespace pattern is easy to learn.
- For experienced developers, commonly repeated imports are boilerplate.

Personal side note: Web2py is very easy to install.

Controversy

Web2py argues practicality
in some very specific places.

Special cases aren't special enough to break the rules.
Although practicality beats purity.

Web2py will always
be contentious

And that's okay

Daniel Greenfeld
pydanny.com / @pydanny

A Little Magic Goes a Long Way

Daniel Greenfeld
pydanny.com / @pydanny

A Little Magic Goes a Long Way

Flask and its global **Request** object

http://bit.ly/flask-requests

Daniel Greenfeld
pydanny.com / @pydanny

Exceptions

Silent
Exceptions
are the
Devil

Exceptions

Errors should never pass silently.
Unless explicitly silenced.*

* Zen of Python, lines 10 and 11

Daniel Greenfeld
pydanny.com / @pydanny

djangopackages.com

- Once a day iterates across all packages.
- Updates the metadata from:
- Github:
- Bitbucket
- PyPI

Django Packages

Problems

- Sometimes the APIs go down.
- Sometimes the APIs change.
- Sometimes projects get deleted.
- Sometimes the Internets fail

Catch and report exceptions!

Old package_updater.py

```
...
for package in Package.objects.all():
    try:
        package.fetch_metadata()
        package.fetch_commits()
    except socket_error, e:
        text += "\nFor '%s', threw a socket_error: %s" % \
                    (package.title, e)
        continue
    # snip lots of other exceptions
    except Exception as e:
        text += "\nFor '%s', General Exception: %s" % \
                    (package.title, e)
        continue

# email later
```

http://bit.ly/Q8v9xk

https://github.com/opencomparison/opencomparison/blob/master/package/management/commands/package_updater.py

Daniel Greenfeld
pydanny.com / @pydanny

What I was doing

(and it's wrong)

```
>>> try:
...     a = b
... except Exception as e:
...     print(e)
...
name 'b' is not defined
```

What's the error type?!?

Where is my stack trace?!?

Daniel Greenfeld
pydanny.com / @pydanny

What I wanted

Traceback

```
>>> a = b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
```

Error type

Error message

Exceptions

My code is nearly silent

Errors should never pass silently.
Unless explicitly silenced.*

I've silenced things for no good reason

* Zen of Python, lines 10 and 11


Daniel Greenfeld
pydanny.com / @pydanny

Getting what I want

```
>>> class CustomErrorHandler(Exception):
...     def __init__(self, error):
...         print(error)
...         print(type(error))
...
>>> try:
...     a=b
... except Exception as e:
...     raise CustomErrorHandler(e)
...
name 'b' is not defined
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
__main__.CustomErrorHandler
NameError
```

For this example print == log

Error message

Traceback

No color because it's a print statement

Error Type

Daniel Greenfeld
pydanny.com / @pydanny

PackageUpdaterException

```
class PackageUpdaterException(Exception):
    def __init__(self, error, title):
        log_message = "For {title}, {error_type}: {error}".format(
            title=title,
            error_type=type(error),
            error=error
        )
        logging.error(log_message)
        logging.exception(error)

for package in Package.objects.all():
    try:
        try:
            package.fetch_metadata()
            package.fetch_commits()
        except Exception as e:
            raise PackageUpdaterException(e, package.title)
    except PackageUpdaterException:
        continue
```

Nice message

Full traceback

All errors caught

Loop forward

Daniel Greenfeld
pydanny.com / @pydanny

Exceptions

My code is nearly silent

Errors should never pass silently.

Unless explicitly silenced.

I've silenced things for no good reason

Daniel Greenfeld
pydanny.com / @pydanny

Exceptions

Errors should never pass silently.
Unless explicitly silenced.


Daniel Greenfeld
pydanny.com / @pydanny

Next

up...


Daniel Greenfeld
pydanny.com / @pydanny

The Dutch Way

Decorators

```
@memoize
def allcaps(string):
    return string.upper()
```

>

```
def allcaps(string):
    return string.upper()


allcaps = memoize(allcaps)
```

Decorators are
easy to explain!

"A decorator is a function that returns a function."

Daniel Greenfeld
pydanny.com / @pydanny

I am Zen

Decorators == Zen of Python


Daniel Greenfeld
pydanny.com / @pydanny

Until...


Daniel Greenfeld
pydanny.com / @pydanny

I am not Zen

I need to write a decorator.


Daniel Greenfeld
pydanny.com / @pydanny

Ouch

You try to shoot yourself in the foot, only to realize there's no need, since Guido thoughtfully shot you in the foot years ago.

-- Nick Mathewson, comp.lang.python

http://starship.python.net/~mwh/quotes.html

Decorators

```
@memoize
def allcaps(string):
    return string.upper()
```

>

```
def allcaps(string):
    return string.upper()


allcaps = memoize(allcaps)
```

Decorators are
easy to explain!

"A decorator is a function that returns a function."

Daniel Greenfeld
pydanny.com / @pydanny

Decorator Template

```
def decorator(function_to_decorate):
    def wrapper(*args, **kwargs):
        # do something before invoation
        result = func_to_decorate(*args, **kwargs)

        # do something after
        return result
    # update wrapper.__doc__ and .func_name
    # or functools.wraps
    return wrapper
```

Wrapper function does things before and after the function is called here.

Result is returned when the wrapper is done

When decorated function is called decorator returns wrapper

http://pydanny-event-notes.readthedocs.org/en/latest/SCALE10x/python-decorators.html#decorator-template


Daniel Greenfeld
pydanny.com / @pydanny

The Dutch Way

There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first **unless you're Dutch**.*

* Zen of Python, lines 13 and 14


Daniel Greenfeld
pydanny.com / @pydanny

Decorator implementation

Datastore

```
def memoize(func):
    cache = {}

    def memoized(*args):
        if args in cache:
            return cache[args]
        result = cache[args] = func(*args)
        return result

    return memoized
```

Return value if args in cache

set cache

Return value

Return function

```
@memoize
def allcaps(string):
    return string.upper()
```

Daniel Greenfeld
pydanny.com / @pydanny

Whew.

Daniel Greenfeld
pydanny.com / @pydanny

What about decorators that accept arguments?


Daniel Greenfeld
pydanny.com / @pydanny

Oh No.

Daniel Greenfeld
pydanny.com / @pydanny

Explaining this is

Hard.

That's because we create a decorator that creates a parameterized function to wrap the function.

Daniel Greenfeld
pydanny.com / @pydanny

multiplier decorator

```
@multiplier(5)
def allcaps(string):
    return string.upper()
```

Multiplier function sets the state for the **multiple** argument

```
def multiplier(multiple):
    def decorator(function):
        def wrapper(*args, **kwargs):
            return function(*args, **kwargs) * multiple
        return wrapper
    return decorator
```

Wrapper function does:

Result is returned when the wrapper is done.

When decorated function is called the decorator function returns the wrapper function

What am I supposed to highlight?

Daniel Greenfeld
pydanny.com / @pydanny

Whew

Daniel Greenfeld
pydanny.com / @pydanny

Oh No.

Daniel Greenfeld
pydanny.com / @pydanny

Not Done Yet!

Daniel Greenfeld
pydanny.com / @pydanny

authentication decorator

```
@authorization('admin')
def do_admin_thing(user):
    # do something administrative
    return user
```

Don't forget functools!

```
import functools
def authorization(roles):
    def decorator(function):
        @functools.wraps(function)
        def wrapper(*args, **kwargs):
            check_roles(user, roles)
            return function(*args, **kwargs)
        return wrapper
    return decorator
```


Daniel Greenfeld
pydanny.com / @pydanny

Whew


Daniel Greenfeld
pydanny.com / @pydanny

Really.

Daniel Greenfeld
pydanny.com / @pydanny

I'm not doing class decorators.

Daniel Greenfeld
pydanny.com / @pydanny

It is not easy to explain how to write decorators.

Daniel Greenfeld
pydanny.com / @pydanny

Contention

While Using decorators is Zen...

Daniel Greenfeld
pydanny.com / @pydanny

Contention

Writing Decorators is Not.


Daniel Greenfeld
pydanny.com / @pydanny

Deep Thought

Decorators are
easy to explain!

There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.

If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.

Although practicality beats purity.

Decorators are
hard to explain!

Daniel Greenfeld
pydanny.com / @pydanny

Use the decorator library

https://pypi.python.org/pypi/decorator


Daniel Greenfeld
pydanny.com / @pydanny

Avoiding Technical Debt

Part I

Getting it done

vs.

Technical debt

Now is better than never.
Although never is often better than *right* now.

* Zen of Python, lines 15 and 16

Daniel Greenfeld
pydanny.com / @pydanny

Some things take time

- Tests
- Documentation

(Risks of skipping them)

Risk: Multiple coding standards

Risk: Deploying broken code

Risk: problems upgrading dependencies

Risk: Forgetting install/deploy

Easy Test Patterns

For developers racing to meet deadlines:

- **Always make sure your test harness can run**
- Try using tests instead of the shell/repl.
- After the first deadline, reject any incoming code that drops coverage.
- Use coverage.py

Daniel Greenfeld
pydanny.com / @pydanny

Must-have Documentation

- Installation/Deployment procedures
- Coding standards
- How to run tests
- Version (including __version__)

Daniel Greenfeld
pydanny.com / @pydanny

Easy Test Patterns

- Always make sure your test harness can run
- Try using tests instead of the shell/repl.
- **Reject any incoming code that drops coverage.**
- **Use coverage.py**

Daniel Greenfeld
pydanny.com / @pydanny

Getting technical again...

Daniel Greenfeld
pydanny.com / @pydanny

Avoiding Technical Debt

Part II

Namespaces

```
import re
import os
from django import forms
from myproject import utils
from twisted.internet import protocol, reactor
```

- Extremely powerful
- Useful
- Precise

Daniel Greenfeld
pydanny.com / @pydanny

import * makes development faster[1]

```
from re import *
from os import *
from twisted import *
from django.forms import *
from myproject.utils import *
```

- Extremely powerful
- Useful
- Imports everything at once! [2]

[1]Warning: import * can be dangerous

[2]Warning: import * can be dangerous

Comparing two modules

```
def compare(mod1, mod2):
    title = '\nComparing {0}, {1}:'.format(
            mod1.__name__,
            mod2.__name__
            )
    print(title)
    for x in dir(mod1):
        for y in dir(mod2):
            if x == y and not x.startswith('_'):
                print("* " + x)
```

Comparing two modules

```
from re import *
from os import *
```

```
>>> import re
>>> import os

>>> compare(os, re)
Comparing os, re:
* sys
* error
```

```
>>> re.sys == os.sys
True

>>> re.error == os.error
False
```

import * can get you into trouble

Daniel Greenfeld
pydanny.com / @pydanny

Breaking built-ins

```
def compare_builtins(mod1):
    print("\nComparing {0} to builtins:".format(mod1.__name__))
    for x in dir(mod1):
        for y in dir(globals()['__builtins__']):
            if x == y and not x.startswith('_'):
                print("* GLOBAL: {0}".format(x))
```

Checks to see if a module has items that match any Python built-in.

Daniel Greenfeld
pydanny.com / @pydanny

Breaking built-ins

```
from re import *
from os import *
```

Compare ‘re’

```
>>> compare_builtins(re)
Comparing re to builtins:
* GLOBAL: compile
```

Breaks compile() built-in.

Annoying but
infrequent problem.

Compare ‘os’

```
>>> compare_builtins(os)
Comparing os to builtins:
* GLOBAL: open
```

Breaks open() built-in.

This can drive you crazy.

Daniel Greenfeld
pydanny.com / @pydanny

The open() story

before

```
Help on built-in function open in module __builtin__:

open(...)
    open(name[, mode[, buffering]]) -> file object

    Open a file using the file() type, returns a file object.  This is the
    preferred way to open a file.  See file.__doc__ for further information.
```

after `from os import *`

```
Help on built-in function open in module posix:

open(...)
    open(filename, flag [, mode=0777]) -> fd

    Open a file (for low level IO).
```

Breaks all the things!

Daniel Greenfeld
pydanny.com / @pydanny

Beginner pro-tip

Be careful of tutorials that use *import **.

Daniel Greenfeld
pydanny.com / @pydanny

Contention

import * is not for beginners.

import * is people who really know Python.

```
__all__ = ["echo", "surround", "reverse"]
```

Daniel Greenfeld
pydanny.com / @pydanny

Summary

Stay this person

Myself at 13 in front of the Apple][

Admit What You Don't Know


Daniel Greenfeld
pydanny.com / @pydanny

Stay out of your comfort Zone

Daniel Greenfeld
pydanny.com / @pydanny

Grow

Daniel Greenfeld
pydanny.com / @pydanny

What I Want To Know

- Twisted
- Numpy
- SciPy
- Tulip
- C
- Etc.


Daniel Greenfeld
pydanny.com / @pydanny

If I continue to Learn

Daniel Greenfeld
pydanny.com / @pydanny

I Get To Be This Person

Daniel Greenfeld
pydanny.com / @pydanny

Think

Hard


Daniel Greenfeld
pydanny.com / @pydanny

Thank you

- Armin Ronacher
- nephila.it
- Richard Jones
- Raymond Hettiger
- EuroPython
- PyKonik
- Łukasz Langa
- Tomasz Paczkowski

Thank you

- Matt Harrison
- Ola Sendecka
- Kenneth Love
- Lennart Regebro
- Paul Hildebrandt
- Audrey Roy


One More Thing...


Daniel Greenfeld
pydanny.com / @pydanny

Finis

Daniel Greenfeld
pydanny.com / @pydanny

Q & A

Daniel Greenfeld
pydanny.com / @pydanny