Thinking Hard About Python
Danny: 128,546++
Audrey: 121,871++

http://2scoops.org
What I want you to think of me.
What I’m really like.

Myself at 13 in front of the Apple
Overview
Coding into Trouble
Controversy
Exceptions
Avoiding Technical Debt
...first section...
Coding into Trouble
a.k.a.
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()
Contention

The super() method can create ambiguity.
Example: Django
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.
However...
Things I don’t know:

The ancestor chain for

django.views.generic.edit.UpdateView
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
```
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?
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)
Ancestor Chain (MRO) of ActionUpdateView

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

Print the MRO

MRO = Method Resolution Order
Ancestor Chain (MRO)

<class 'actions.views.ActionUpdateView'>
<class 'braces.views.LoginRequiredMixin'>
<class 'actions.views.ActionBaseView'>
<class 'core.views.AuthorizedForProtocolMixim'>
<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

```python
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() method
Ancestor Chain (MRO) of

Current class

```
<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'>
```

super’s chosen form_valid() ancestor
Whew!
Safe!
If you’re not careful, super can cause subtle inheritance/MRO problems.
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.
Controversy
Special cases aren’t special enough to break the rules.

Although practicality beats purity.*

* Zen of Python, lines 8 and 9
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!
Special cases aren’t special enough to break the rules.

Although practicality beats purity.*

* Zen of Python, lines 8 and 9
Web2py

Often honors Implicit over Explicit

Follows its own namespace pattern
Web2py code sample

```python
# 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
```
Web2py code sample

I GET IT NOW

Europe taught me why unicode is important.
Web2py code sample

```python
# 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

def handle_request(request):
    pass

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
```
Web2py code sample

```python
# 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 limits.
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
```

Response object magically exists. No import necessary

Written by Massimo himself

What about namespace pollution?

What can I expect in any location?
Contestation

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
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
A Little Magic Goes a Long Way
A Little Magic Goes a Long Way

Flask and its global Request object

Exceptions
Silent Exceptions are the Devil
Exceptions

Errors should never pass silently. Unless explicitly silenced.*

* Zen of Python, lines 10 and 11
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

```python
for package in Package.objects.all():
    try:
        package.fetch_metadata()
        package.fetch_commits()
    except socket_error, e:
        text += '
        For `{title}`, threw a socket_error: `{e}`
        '.format(package=package, title=package.title, e=e)
        continue
    # snip lots of other exceptions
    except Exception as e:
        text += '
        For `{title}`, General Exception: `{e}`
        '.format(package=package, title=package.title, e=e)
        continue

# email later


```
What I was doing

(and it’s wrong)

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

What’s the error type?!?

Where is my stack trace?!?
What I wanted

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

Errors should never pass silently. Unless explicitly silenced.*

* Zen of Python, lines 10 and 11
Getting what I want

```python
>>> 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

No color because it's a print statement

Error Type
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
Loop forward

Nice message
Full traceback
All errors caught
Exceptions

Errors should never pass silently. Unless explicitly silenced.

My code is nearly silent

I’ve silenced things for no good reason
Exceptions

Errors should never pass silently. Unless explicitly silenced.
Next
up...
The Dutch Way
Decorators

Decorators are easy to explain!

“"A decorator is a function that returns a function."
I am Zen

Decorators == Zen of Python
Until...
I am not Zen

I need to write a decorator.
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 are easy to explain!

“A decorator is a function that returns a function.”
Decorator Template

```python
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
```

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
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
def memoize(func):
    cache = {}

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

@memoize
def allcaps(string):
    return string.upper()
Whew.
What about decorators that accept arguments?
Oh No.
Explaining this is Hard.

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

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

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

@multiplier(5)
def allcaps(string):
    return string.upper()
Whew
Oh No.
Not Done Yet!
**authentication decorator**

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

```python
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
```

Don’t forget functools!
Whew
Really.
I’m not doing class decorators.
It is not easy to explain how to write decorators.
Contention

While Using decorators is Zen...
Contention

Writing Decorators is Not.
Deep Thought

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 easy to explain!

Decorators are hard to explain!
Use the decorator library

https://pypi.python.org/pypi/decorator
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
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
Must-have Documentation

• Installation/Deployment procedures
• Coding standards
• How to run tests
• Version (including __version__)
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
Getting technical again...
Avoiding Technical Debt

Part II
Namespaces

- Extremely powerful
- Useful
- Precise
import * makes development faster

• 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 = '
Comparing {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

```python
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
Breaking built-ins

```python
def compare_builtins(mod1):
    print("Comparing \{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.
Breaking built-ins

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

**Compare ‘re’**

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

Breaks `compile()` built-in.

Annoying but infrequent problem.

**Compare ‘os’**

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

Breaks `open()` built-in.

This can drive you crazy.
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!
Beginner pro-tip

Be careful of tutorials that use import *. 
Contention

import * is not for beginners.

import * is people who really know Python.

__all__ = ["echo", "surround", "reverse"]
Stay this person

Myself at 13 in front of the Apple
Admit What You Don't Know
Stay out of your comfort Zone
Grow
What I Want To Know

• Twisted
• Numpy
• SciPy

• Tulip
• C
• Etc.
If I continue to Learn
I Get To Be This Person
Think Hard
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...
Finis
Q & A