Refactoring Python: Why and how to restructure your code

Brett Slatkin
@haxor
onebigfluke.com
Agenda

- What, When, Why, How
- Strategies
  - Extract Variable & Function
  - Extract Class & Move Fields
  - Move Field gotchas
- Follow-up
- Bonus
  - Extract Closure
What is refactoring?

Repeatedly reorganizing and rewriting code until it's obvious* to a new reader.

* See Clean Code by Robert Martin
When do you refactor?

- In advance
- For testing
- "Don't repeat yourself"
- Brittleness
- Complexity
What's the difference between good and great programmers? (anecdotally)

- **Me**: Usually
  - Writing & testing: 100%
  - Refactoring: 0%
  - Style & docs: 0%

- **Good**
  - Writing & testing: 80%
  - Refactoring: 20%
  - Style & docs: 0%

- **Great**
  - Writing & testing: 70%
  - Refactoring: 30%
  - Style & docs: 0%
How do you refactor?

1. Identify bad code
2. Improve it
3. Run tests
4. Fix and improve tests
5. Repeat
How do you refactor in practice?

- Rename, split, move
- Simplify
- Redraw boundaries
The canonical reference (1999)
But...
But... it's for Java programmers
The more recent version (2009)
But...
But... it's for Ruby programmers
Strategies
Prerequisites

- Thorough tests
- Quick tests
- Source control
- Willing to make mistakes
Extract Variable &
Extract Function
When should you eat certain foods?

MONTHS = ('January', 'February', ...)

def what_to_eat(month):
    if (month.lower().endswith('r') or month.lower().endswith('ary')):
        print('%s: oysters' % month)
    elif 8 > MONTHS.index(month) > 4:
        print('%s: tomatoes' % month)
    else:
        print('%s: asparagus' % month)
When should you eat certain foods?

what_to_eat('November')
what_to_eat('July')
what_to_eat('March')

>>> November: oysters
    July: tomatoes
    March: asparagus
if (month.lower().endswith('r') or month.lower().endswith('ary')):
    print('%s: oysters' % month)
elif 8 > MONTHS.index(month) > 4:
    print('%s: tomatoes' % month)
else:
    print('%s: asparagus' % month)
After: Extract variables

```python
lowered = month.lower()
ends_in_r = lowered.endswith('r')
ends_in_ary = lowered.endswith('ary')
index = MONTHS.index(month)
summer = 8 > index > 4

if ends_in_r or ends_in_ary:
    print('%s: oysters' % month)
elif summer:
    print('%s: tomatoes' % month)
else:
    print('%s: asparagus' % month)
```
Extract variables into functions

def oysters_good(month):
    lowered = month.lower()
    return (lowered.endswith('r') or lowered.endswith('ary'))

def tomatoes_good(month):
    index = MONTHS.index(month)
    return 8 > index > 4
Before

```python
if (month.lower().endswith('r') or month.lower().endswith('ary')):
    print('%s: oysters' % month)
elif 8 > MONTHS.index(month) > 4:
    print('%s: tomatoes' % month)
else:
    print('%s: asparagus' % month)
```
After: Using functions

```python
if oysters_good(month):
    print('%s: oysters' % month)
elif tomatoes_good(month):
    print('%s: tomatoes' % month)
else:
    print('%s: asparagus' % month)
```
After: Using functions with variables

time_for_oysters = oysters_good(month)
time_for_tomatoes = tomatoes_good(month)

if time_for_oysters:
    print('%s: oysters' % month)
elif time_for_tomatoes:
    print('%s: tomatoes' % month)
else:
    print('%s: asparagus' % month)
def oysters_good(month):
    lowered = month.lower()
    return (lowered.endswith('r') or lowered.endswith('ary'))

def tomatoes_good(month):
    index = MONTHS.index(month)
    return 8 > index > 4
Extract variables into classes

class OystersGood:
    def __init__(self, month):
        lowered = month.lower()
        self.r = lowered.endswith('r')
        self.ary = lowered.endswith('ary')
        self._result = self.r or self.ary
    
    def __bool__(self):  # aka __nonzero__
        return self._result
Extract variables into classes

class TomatoesGood:
    def __init__(self, month):
        self.index = MONTHS.index(month)
        self._result = 8 > index > 4

    def __bool__(self): # aka __nonzero__
        return self._result
Before: Using functions

time_for_oysters = oysters_good(month)
time_for_tomatoes = tomatoes_good(month)

if time_for_oysters:
    print('%s: oysters' % month)
elif time_for_tomatoes:
    print('%s: tomatoes' % month)
else:
    print('%s: asparagus' % month)
After: Using classes

time_for_oysters = OystersGood(month)
time_for_tomatoes = TomatoesGood(month)

if time_for_oysters:  # Calls __bool__
    print('%s: oysters' % month)
elif time_for_tomatoes:  # Calls __bool__
    print('%s: tomatoes' % month)
else:
    print('%s: asparagus' % month)
Extracting classes facilitates testing

test = OystersGood('November')
assert test
assert test.r
assert not test.ary

test = OystersGood('July')
assert not test
assert not test.r
assert not test.ary
Things to remember

● Extract variables and functions to improve readability

● Extract variables into classes to improve testability

● Use __bool__ to indicate a class is a paper trail
Extract Class &
Move Fields
Keeping track of your pets

class Pet:
    def __init__(self, name):
        self.name = name
Keeping track of your pets

```python
pet = Pet('Gregory the Gila')
print(pet.name)
```

```>>> Gregory the Gila```
Keeping track of your pet's age

class Pet:
    def __init__(self, name, age):
        self.name = name
        self.age = age
Keeping track of your pet's age

code:

```python
pet = Pet('Gregory the Gila', 3)

print('%s is %d years old' % (pet.name, pet.age))
```

output:

```
>>> Gregory the Gila is 3 years old
```
class Pet:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.treats_eaten = 0

    def give_treats(self, count):
        self.treats_eaten += count

Keeping track of your pet's treats
Keeping track of your pet's treats

```python
pet = Pet('Gregory the Gila', 3)

pet.give_treats(2)

print('%s ate %d treats' %
      (pet.name, pet.treats_eaten))

>>> Gregory the Gila ate 2 treats
```
Keeping track of your pet's needs

class Pet:
    def __init__(self, name, age, *,
                 has_scales=False,
                 lays_eggs=False,
                 drinks_milk=False):
        self.name = name
        self.age = age
        self.treats_eaten = 0
        self.has_scales = has_scales
        self.lays_eggs = lays_eggs
        self.drinks_milk = drinks_milk
Keeping track of your pet's needs

class Pet:
    def __init__(self, ...): ...

    def give_treats(self, count): ..

@property
def needs_heat_lamp(self):
    return (
        self.has_scales and
        self.lays_eggs and
        not self.drinks_milk)
Keeping track of your pet's needs

```python
pet = Pet('Gregory the Gila', 3,
          has_scales=True,
          lays_eggs=True)

print('%s needs a heat lamp? %s'
      (pet.name, pet.needs_heat_lamp))

>>> Gregory the Gila needs a heat lamp? True
```
It's getting complicated

class Pet:
    def __init__(self, name, age, *, 
                   has_scales=False, 
                   lays_eggs=False, 
                   drinks_milk=False):
        self.name = name
        self.age = age
        self.treats_eaten = 0
        self.has_scales = has_scales
        self.lays_eggs = lays_eggs
        self.drinks_milk = drinks_milk
How do you redraw boundaries?

1. Add an improved interface
   ○ Maintain backwards compatibility
   ○ Issue warnings for old usage

2. Migrate old usage to new usage
   ○ Run tests to verify correctness
   ○ Fix and improve broken tests

3. Remove code for old interface
What are warnings?

```python
import warnings
warnings.warn('Helpful message')
```

- Default: Print messages to stderr
- Force warnings to become exceptions:
  
  ```bash
  python -W error your_code.py
  ```
class Pet:
    def __init__(self, name, age, *,
                 has_scales=False,
                 lays_eggs=False,
                 drinks_milk=False):
        self.name = name
        self.age = age
        self.treats_eaten = 0
        self.has_scales = has_scales
        self.lays_eggs = lays_eggs
        self.drinks_milk = drinks_milk
After: Extract Animal from Pet

class Animal:
    def __init__(self, *,
        has_scales=False,
        lays_eggs=False,
        drinks_milk=False):

    self.has_scales = has_scales
    self.lays_eggs = lays_eggs
    self.drinks_milk = drinks_milk
class Pet:
    def __init__(self, name, age, *,
                  has_scales=False, lays_eggs=False, drinks_milk=False):
        ...

After: Add / intro parameter object

class Pet:
    def __init__(self, name, age,
                 animal=None, **kwargs):
        ...

class Pet:
    def __init__(self, name, age, animal=None, **kwargs):
        if kwargs and animal is not None:
            raise TypeError('Mixed usage')
        if animal is None:
            warnings.warn('Should use Animal')
            animal = Animal(**kwargs)
        self.animal = animal
        self.name = name
        self.age = age
        self.treats_eaten = 0
Mixed usage raises exception

animal = Animal(has_scales=True,
                lays_eggs=True)

pet = Pet('Gregory the Gila', 3,
         animal, has_scales=False)

>>> Traceback ... 
TypeError: Mixed usage
Old constructor works, but warns

```
> pet = Pet('Gregory the Gila', 3,
>            has_scales=True,
>            lays_eggs=True)

>>> UserWarning: Should use Animal
New constructor usage doesn't warn

```python
animal = Animal(has_scales=True,
                lays_eggs=True)
pet = Pet('Gregory the Gila', 3, animal)

print('My pet is %s' % pet.name)
```

```plaintext
>>> My pet is Gregory the Gila
```
class Pet:
    def __init__(self, name, age, *,
        has_scales=False,
        lays_eggs=False,
        drinks_milk=False):
        ...
        self.has_scales = has_scales
        self.lays_eggs = lays_eggs
        self.drinks_milk = drinks_milk
After: Move fields to inner object

class Pet:
    ...
    @property
def has_scales(self):
        warnings.warn('Use animal attribute')
        return self.animal.animal.has_scales

@property
def lays_eggs(self): ...

@property
def drinks_milk(self): ...
Old attributes issue a warning

```python
animal = Animal(has_scales=True, lays_eggs=True)
pet = Pet('Gregory the Gila', 3, animal)

print('%s has scales? %s' % (pet.name, pet.has_scales))
```

```pythonhistory
>>> UserWarning: Use animal attribute Gregory the Gila has scales? True
```
New attributes don't warn

animal = Animal(has_scales=True, 
                lays_eggs=True)

pet = Pet('Gregory the Gila', 3, animal)

print('%s has scales? %s' % 
      (pet.name, pet.animal.animal.has_scales))

>>> Gregory the Gila has scales? True
Before: Helpers access self

class Pet:
    def __init__(self, ...): ...

    def give_treats(self, count): ..

@property
def needs_heat_lamp(self):
    return (self.has_scales and
            self.lays_eggs and
            not self.drinks_milk)
After: Helpers access inner object

class Pet:
    def __init__(self, ...): ...

    def give_treats(self, count): ..

@property
    def needs_heat_lamp(self):
        return (self.animal.has_scales and self.animal.lays_eggs and not self.animal.drinks_milk)
Existing helper usage doesn't warn

animal = Animal(has_scales=True, 
                lays_eggs=True)

pet = Pet('Gregory the Gila', 3, animal)

print('%s needs a heat lamp? %s' %
      (pet.name, pet.needs_heat_lamp))

>>> Gregory the Gila needs a heat lamp? True
Things to remember

- Split classes using optional arguments to `__init__`

- Use `@property` to move methods and fields between classes

- Issue warnings in old code paths to find their occurrences
Move Field gotchas
Before: Is this obvious?

class Animal:
    def __init__(self, *,
        has_scales=False,
        lays_eggs=False,
        drinks_milk=False):
        ...

class Pet:
    def __init__(self, name, age, animal):
        ...

class Animal:
    def __init__(self, age=None, *,
                 has_scales=False,
                 lays_eggs=False,
                 drinks_milk=False):
        ...

class Pet:
    def __init__(self, name, animal):
        ...

After: Move age to Animal
After: Constructor with optional age

class Animal:
    def __init__(self, age=None, *,
                has_scales=False,
                lays_eggs=False,
                drinks_milk=False):
        if age is None:
            warnings.warn('age not specified')
        self.age = age
        self.has_scales = has_scales
        self.lays_eggs = lays_eggs
        self.drinks_milk = drinks_milk
Before: Pet constructor with age

class Pet:
    def __init__(self, name, age, animal):
        ...

class Pet:
    def __init__(self, name, maybe_age,
                 maybe_animal=None):
        ...

class Pet:
    def __init__(self, name, maybe_age,
                 maybe_animal=None):
        if maybe_animal is not None:
            warnings.warn('Put age on animal')
            self.animal = maybe_animal
            self.animal.age = maybe_age
        else:
            self.animal = maybe_age

...
After: Compatibility property `age`

```python
class Pet:
    def __init__(self, name, maybe_age, maybe_animal=None): ...

    def give_treats(self, count): ...

@property
def age(self):
    warnings.warn('Use animal.age')
    return self.animal.age
```
After: Old usage has a lot of warnings

```python
animal = Animal(has_scales=True, lays_eggs=True)

pet = Pet('Gregory the Gila', 3, animal)

print('%s is %d years old' % (pet.name, pet.age))
```

```python
>>> UserWarning: age not specified
UserWarning: Put age on animal
UserWarning: Use animal.age
Gregory the Gila is 3 years old
```
After: New usage has no warnings

```python
animal = Animal(3, has_scales=True, lays_eggs=True)
pet = Pet('Gregory the Gila', animal)

print('%s is %d years old' %
      (pet.name, pet.animal.age))

>>> Gregory the Gila is 3 years old
```
Gregory is older than I thought

```python
animal = Animal(3, has_scales=True, lays_eggs=True)
pet = Pet('Gregory the Gila', animal)
pet.age = 5
```
Assigning to age breaks!

```python
animal = Animal(3, has_scales=True, lays_eggs=True)
pet = Pet('Gregory the Gila', animal)
pet.age = 5  # Error
```

```
>>> AttributeError: can't set attribute
```
Need a compatibility property setter

class Pet:
    ...

@property
def age(self):
    warnings.warn('Use animal.age')
    return self.animal.age

@age.setter
def age(self, new_age):
    warnings.warn('Assign animal.age')
    self.animal.age = new_age
Old assignment now issues a warning

animal = Animal(3, has_scales=True, lays_eggs=True)
pet = Pet('Gregory the Gila', animal)
pet.age = 5

>>> UserWarning: Assign animal.age
New assignment doesn't warn

animal = Animal(3, has_scales=True, lays_eggs=True)
pet = Pet('Gregory the Gila', animal)
pet.animal.age = 5

print('%s is %d years old' % (pet.name, pet.animal.age))

>>> Gregory the Gila is 5 years old
Finally: age is part of Animal

class Animal:
    def __init__(self, age, *,
        has_scales=False,
        lays_eggs=False,
        drinks_milk=False):
        self.age = age
        self.has_scales = has_scales
        self.lays_eggs = lays_eggs
        self.drinks_milk = drinks_milk

...
Finally: Pet has no concept of age

class Pet:
    def __init__(self, name, animal):
        self.animal = animal
        self.name = name
        self.treats_eaten = 0

...
Again: Gregory is older than I thought

animal = Animal(3, has_scales=True, lays_eggs=True)
pet = Pet('Gregory the Gila', animal)
pet.age = 5

print('%s is %d years old' % (pet.name, pet.animal.age))
Surprise! Old usage is doubly broken

animal = Animal(3, has_scales=True, lays_eggs=True)
pet = Pet('Gregory the Gila', animal)
pet.age = 5

print('%s is %d years old' % (pet.name, pet.animal.age))

>>> Gregory the Gila is 3 years old
Surprise! Old usage is doubly broken

```python
animal = Animal(3, has_scales=True, lays_eggs=True)
pet = Pet('Gregory the Gila', animal)
pet.age = 5  # No error!

print('%s is %d years old' % (pet.name, pet.animal.age))

>>> Gregory the Gila is 3 years old
```
class Pet:
    ...

@property
def age(self):
    raise AttributeError('Use animal')

@age.setter
def age(self, new_age):
    raise AttributeError('Use animal')
Now accidental old usage will break

```python
animal = Animal(3, has_scales=True, lays_eggs=True)
pet = Pet('Gregory the Gila', animal)
pet.age = 5 # Error

>>> 
Traceback ... 
AttributeError: Use animal
Things to remember

- Use `@property.setter` to move fields that can be assigned

- Defend against muscle memory with tombstone `@property`s
Follow-up
Links

- **PMOTW: Warnings** - Doug Hellmann
- **Stop Writing Classes** - Jack Diederich
- **Beyond PEP 8** - Raymond Hettinger
Links

• This talk's code & slides:
  ○ github.com/bslatkin/pycon2016

• My book: EffectivePython.com
  ○ Discount today: informit.com/deals

• Me: @haxor and onebigfluke.com
Appendix
Bonus: Extract Closure
Calculating stats for students

class Grade:
    def __init__(self, student, score):
        self.student = student
        self.score = score

grades = [
    Grade('Jim', 92), Grade('Jen', 89),
    Grade('Ali', 73), Grade('Bob', 96),
]
Calculating stats for students

def print_stats(grades):
    total, count, lo, hi = 0, 0, 100, 0
    for grade in grades:
        total += grade.score
        count += 1
        if grade.score < lo:
            lo = grade.score
        elif grade.score > hi:
            hi = grade.score
    print('Avg: %f, Lo: %f Hi: %f' % (total / count, lo, hi))
Calculating stats for students

print_stats(grades)

>>> Avg: 87.5, Lo: 73.0, Hi: 96.0
def print_stats(grades):
    total, count, lo, hi = 0, 0, 100, 0
    for grade in grades:
        total += grade.score
        count += 1
        if grade.score < lo:
            lo = grade.score
        elif grade.score > hi:
            hi = grade.score
    print('Avg: %f, Lo: %f Hi: %f' % (total / count, lo, hi))
def print_stats(grades):
    total, count, lo, hi = 0, 0, 100, 0

def adjust_stats(grade):
    # Closure
    ...

for grade in grades:
    adjust_stats(grade)

print('Avg: %f, Lo: %f Hi: %f' % (total / count, lo, hi))
Stateful closure functions are messy

def print_stats(grades):
    total, count, lo, hi = 0, 0, 100, 0

    def adjust_stats(grade):
        nonlocal total, count, lo, hi
        total += grade.score
        count += 1
        if grade.score < lo:
            lo = grade.score
        elif grade.score > hi:
            hi = grade.score

    ...
Instead: Stateful closure class

class CalculateStats:
    def __init__(self):
        self.total = 0
        self.count = 0
        self.lo = 100
        self.hi = 0

    def __call__(self, grade): ...

@property
def avg(self): ...
Instead: Stateful closure class

class CalculateStats:
    def __init__(self): ...

    def __call__(self, grade):
        self.total += grade.score
        self.count += 1
        if grade.score < self.lo:
            self.lo = grade.score
        elif grade.score > self.hi:
            self.hi = grade.score
Instead: Stateful closure class

class CalculateStats:
    def __init__(self): ...

    def __call__(self, grade): ...

@property
def avg(self):
    return self.total / self.count
Before

```python
def print_stats(grades):
    total, count, lo, hi = 0, 0, 100, 0
    for grade in grades:
        total += grade.score
        count += 1
        if grade.score < lo:
            lo = grade.score
        elif grade.score > hi:
            hi = grade.score
    print('Avg: %f, Lo: %f Hi: %f' % (total / count, lo, hi))
```
def print_stats(grades):
    total, count, lo, hi = 0, 0, 100, 0

    def adjust_stats(grade):
        ...

    for grade in grades:
        adjust_stats(grade)

    print('Avg: %f, Lo: %f Hi: %f' % (total / count, lo, hi))
After: Using stateful closure class

def print_stats(grades):
    stats = CalculateStats()

    for grade in grades:
        stats(grade)

    print('Avg: %f, Lo: %f Hi: %f' %
          (stats.avg, stats.lo, stats.hi))
Things to remember

- Extracting a closure function can make code less clear
- Use `__call__` to indicate that a class is just a stateful closure
- Closure classes can be tested independently