Washing away code smells

@yennycheung #EuroPython
ABOUT ME

Yenny Cheung
Originally from Hong Kong

Software Engineer at Yelp
On the Biz National team
In Hamburg

Speaker at PyConDE, PyDays Vienna and Talk Python podcast

@yennychuang #EuroPython
Connecting people with great local businesses
Your Next Review Awaits

- **China Restaurant Golden**
  - Worthens: 4
  - 5 stars

- **Flowers' Whisper**
  - Mitteckengasse: 7
  - 3 stars

- **Ume no Hana**
  - Thudernstr: 15
  - 4 stars

- **Shiso Burger**
  - Bugenhagerstr. 23
  - 4 stars

- **China Fang**
  - Nies 1
  - 4 stars

- **TrüffelSchwein**
  - Mühlenkamp 54
  - 5 stars
Some fun facts about Yelp

🌟 Yelpers have written 155 million reviews since 2004.

💰 We have 74 million desktop and 30 million mobile app monthly unique visitors.

💻 We have over 500 developers.

🛠️ We have over 300 services and our monolith yelp-main has over 3 million lines of code!
Agenda for today

⊙ What are code smells?

❓ Why do we care?

🔍 How to use refactoring to wash away code smells?

🛠️ Tips for bringing refactoring to your company
What are code smells?
"A code smell is a **surface indication** that usually corresponds to a **deeper problem** in the system."

- *Martin Fowler, author of the book “Refactoring”*
Why do we care?
"Let 1,000 flowers bloom. Then rip 999 of them out by the roots."

- Peter Seibel, tech lead for Twitter’s Engineering Effectiveness group
WHY DO WE CARE ABOUT CODE SMELLS

Code smells when left unchecked...

- Builds up tech debt
- Allows code rot!
- Makes it harder to build flexible software
- Decreases productivity and developer happiness

@yennycheung #EuroPython
Refactoring comes to our rescue
Refactoring is a changes the design of your code but not the functionality.
def get_cheese(mood, hunger, money):
    if mood > 3:
        if money == 0:
            return None
        # good mood and hungry
        if hunger > 4:
            return 'bleu'
        # good mood and not hungry
        else:
            return 'american'
    else:
        if mood > 4:
            return None
        if money == 0:
            return None
        else:
            # bad mood and hungry
            if hunger > 4:
                return 'brie'
            # bad mood and not hungry
            else:
                return 'mozzarella'

if __name__ == '__main__':
    cheese = get_cheese(3, 5, 1)
def get_cheese(mood, hunger, money):
    """Evaluate criteria and pick cheese."""

    is_happy = mood > 3
    is_hungry = hunger > 4
    has_money = money > 0

    if not has_money:
        return None

    if is_hungry and not is_happy:
        return 'brie'

    if not is_hungry and is_happy:
        return 'american'

    if not is_hungry and not is_happy:
        return 'mozzarella'

    else:
        return 'bleu'

if __name__ == '__main__':
    cheese = get_cheese(
        mood=3,
        hunger=5,
        money=1,
    )
Refactoring
deep-dive
def get_cheese(mood, hunger, money):
    if mood > 3:
        if money == 0:
            return None
        # good mood and hungry
        if hunger > 4:
            return 'bleu'
        # good mood and not hungry
        else:
            return 'american'
    else:
        if mood > 4:
            return None
        if money == 0:
            return None
        else:
            # bad mood and hungry
            if hunger > 4:
                return 'brie'
            # bad mood and not hungry
            else:
                return 'mozzarella'

if __name__ == '__main__':
    cheese = get_cheese(3, 5, 1)
Refactoring Techniques

- Name it right!
- Get organized
- Picking the right data structure
Name it **right!**

A cure for uncommunicative naming

- Python is **dynamically typed**

- Variable, function & **module** naming

- **Keyword arguments** increase clarity

- Replace magic strings and numbers with ** Enums!**
Enum how to

Explicit

Supports iterable

Enum members are hashable

```python
>>> class Mood(Enum):
...     EXUBERANT = 0
...     CONTENT = 1
...     APATHETIC = 2
...     MELANCHOLIC = 3
...
>>> for mood in Mood:
...     print(mood)
...
Mood.EXUBERANT
Mood.CONTENT
Mood.APATHETIC
Mood.MELANCHOLIC

>>> print(Mood.EXUBERANT)
Mood.EXUBERANT

>>> print(repr(Mood.EXUBERANT))
<Mood.EXUBERANT: 0>

>>> my_mood_count_this_week = {}

>>> my_mood_count_this_week[Mood.EXUBERANT] = 3
>>> my_mood_count_this_week[Mood.MELANCHOLIC] = 1
>>> my_mood_count_this_week[Mood.APATHETIC] = 3

>>> my_mood_count_this_week
{<Mood.APATHETIC: 2>: 3, <Mood.EXUBERANT: 0>: 3,
 <Mood.MELANCHOLIC: 3>: 1}
```
Get organized

A cure for long functions, classes and parameter lists

- Single Responsibility principle
- Function extraction
- Decompose conditionals
- DRY (Don’t repeat yourself)
Fixing long parameter lists

- Long param list
- NamedTuples to the rescue

```python
def identify_cheese(
country,
smell,
touch,
city,
year,
taste,
):
...

class CheeseProductionInfo(NamedTuple):
country: str
city: str
year: str
class CheeseAttributes(NamedTuple):
smell: str
taste: str
touch: str

def identify_cheese(
cheese_production_info,
cheese_attributes,
):
...
```
Picking the right data structure

- Dictionaries
- NamedTuples
- Lists
- Sets
Picking the right data structure

Using dictionaries

Beware that dictionaries are mutables!

```python
>>> def sum_cheese(
    ...     cheese_counts={
    ...         'bleu': 0,
    ...         'brie': 0
    ...     }
    ... ):
    ...     cheese_counts['bleu'] += 1
    ...     return cheese_counts

...    sum_cheese.__defaults__(
    ...        ({'brie': 0, 'bleu': 0},
    ...        )
    ... )

>>> sum_cheese()
    {'bleu': 1, 'brie': 0}

>>> sum_cheese.__defaults__(
    ...        ({'brie': 0, 'bleu': 1},
    ...        )
    ... )
```
from typing import NamedTuple

class CheeseCounts(NamedTuple):
    ...  bleu: int
    ...  brie: int

CheeseCounts.__new__._defaults_ = (0, 0)

print(CheeseCounts(brie=2))
CheeseCounts(bleu=0, brie=2)

print(CheeseCounts())
CheeseCounts(bleu=0, brie=0)
def select_favorite_cheese_from_catalog(cheese_catalog, my_favorite_cheese):
    selected_cheese = []
    for cheese in cheese_catalog:
        if cheese in my_favorite_cheese:
            selected_cheese.append(cheese)
    return selected_cheese

select_favorite_cheese_from_catalog(
    cheese_catalog=[Cheese.BLEU, Cheese.CHEEDDAR],
    my_favorite_cheese=[Cheese.TRUFFLE_BRIE, Cheese.BLEU],
)

>>> [Cheese.BLEU: 'Bleu']
def select_favorite_cheese_from_catalog(cheese_catalog, my_favorite_cheese):
    return cheese_catalog.intersection(my_favorite_cheese)

cheese_catalog = set([Cheese.BLEU, Cheese.CHEDDAR, Cheese.TRUFFLE_BRIE, Cheese.BLEU, Cheese.BLEU]),
my_favorite_cheese = set([Cheese.TRUFFLE_BRIE, Cheese.BLEU, Cheese.BLEU]),

>>> {<Cheese.BLEU: 'Bleu'>}
Check out the standard library, especially `itertools` and `collections` for handy tools!
Testing in the refactoring process

@yennycheung #EuroPython
Testing in the refactoring process

1. Write integration / end-to-end tests for the code to be refactored
   Tests that your application will still behave the same

2. Refactoring

3. Write unit tests for refactored code
   Tests that the code is correct
Tips for **bringing refactoring** to your company
The secret weapon of code reviews

- Boy scout rule: leave it cleaner than you found it
- Encourage refactoring when we add code and fix bugs
How to convince your product manager

- Break down the tasks and take maintenance into account, with refactoring, 4 weeks, otherwise 6 weeks
- If all things fail, abstracting out the implementation detail, adjust estimates to include refactoring and test, “this feature takes X”
Automate your refactoring process

- Yelp’s open source tool: **Undebt**
  Based on pyparsing, massive find and replace tool

- Yelp uses a debt tracker: **Branch Debt**
  Example metrics: noqa count, deprecated function count, lines added to our monolith yelp-main
Takeaways from the talk

What are code smells?

Why do we care?

How to use refactoring to wash away code smells?

Tips for bringing refactoring to your company
We're Hiring!
www.yelp.com/careers/

Offices @
Hamburg
London
San Francisco
Thank you!
Questions?