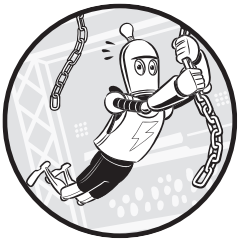
5

Поиск запахов в коде



Если выполнение кода приводит к аварийному завершению программы, с кодом очевидно что-то не так, но сбой — не единственный признак проблем в программах. Другие признаки могут свидетельствовать о наличии более коварных ошибок или неудобочитаемого кода. Подобно тому как запах газа может указывать на утечку, а запах дыма — на возможный пожар, запах кода представляет собой паттерн исходного кода, сигнализирующий о возможных ошибках. Наличие запаха кода не всегда означает, что проблема существует, но вам стоит по крайней мере проанализировать свою программу.

В этой главе я расскажу о некоторых распространенных признаках, указывающих на проблемы в коде. На предотвращение ошибок требуется намного меньше времени и усилий, чем на их выявление, анализ и исправление. У каждого программиста найдется история о том, как он долгие часы отлаживал программу, а потом оказалось, что достаточно было исправить всего одну строку. Из-за этого даже при мимолетном подозрении на потенциальную ошибку вы должны остановиться и лишний раз удостовериться, что вы не создаете себе проблемы в будущем.

Конечно, запахи кода не всегда свидетельствуют о наличии ошибки. В конечном итоге именно вам решать, разбираться со своими подозрениями или не обращать на них внимания.

Дублирование кода

Самый распространенный источник ошибок — *дублирование кода*. Так называют любой исходный код, который создают копированием и вставкой фрагмента вашей программы. Например, следующая короткая программа содержит дублирующийся

код. Обратите внимание: она трижды задает пользователю вопрос о том, как тот себя чувствует (“How are you feeling?”):

```
print('Good morning!')
print('How are you feeling?')
feeling = input()
print('I am happy to hear that you are feeling ' + feeling + '.')
print('Good afternoon!')
print('How are you feeling?')
feeling = input()
print('I am happy to hear that you are feeling ' + feeling + '.')
print('Good evening!')
print('How are you feeling?')
feeling = input()
print('I am happy to hear that you are feeling ' + feeling + '.')
```

Дублирование кода создает проблемы, потому что оно усложняет редактирование кода: изменение в одной копии должно быть внесено во все его дубли в программе. Если вы забудете где-то внести изменение или вы внесете разные изменения в разных копиях, в вашей программе, скорее всего, возникнет ошибка.

Проблема дублирования кода решается *дедупликацией* — код преобразуют так, чтобы он встречался только в одном месте программы — обычно в функции или цикле. В следующем примере я переместил дубликат в функцию, а затем повторно вызывал эту функцию:

```
def askFeeling():
    print('How are you feeling?')
    feeling = input()
    print('I am happy to hear that you are feeling ' + feeling + '.')

print('Good morning!')
askFeeling()
print('Good afternoon!')
askFeeling()
print('Good evening!')
askFeeling()
```

В следующем примере дублирующийся код был перемещен в цикл:

```
for timeOfDay in ['morning', 'afternoon', 'evening']:
    print('Good ' + timeOfDay + '!')
    print('How are you feeling?')
    feeling = input()
    print('I am happy to hear that you are feeling ' + feeling + '.')
```

Также можно объединить два приема и совместить функцию с циклом:

```
def askFeeling(timeOfDay):
    print('Good ' + timeOfDay + '!')
```

```
    print('How are you feeling?')
    feeling = input()
    print('I am happy to hear that you are feeling ' + feeling + '.')

for timeOfDay in ['morning', 'afternoon', 'evening']:
    askFeeling(timeOfDay)
```

Обратите внимание: код, который выдает сообщения «Good morning/afternoon/evening!», похож, но не идентичен. В третьей версии программы я параметризовал код, чтобы исключить дублирование идентичных частей.

Параметр `timeOfDay` и переменная цикла `timeOfDay` заменяют различающиеся части. Теперь, когда из кода были устранены дубликаты (лишние копии), необходимые изменения достаточно внести только в одном месте.

Как и со всеми запахами кода, исключение дублирования не является неукоснительным правилом, которое следует соблюдать всегда. Как правило, чем длиннее фрагмент дубля или чем больше копий присутствует в вашей программе, тем серьезнее стоит задуматься об их устранении. Я не против того, чтобы скопировать код один и даже два раза. Но если в программе присутствуют три или четыре дубля, я обычно серьезно задумываюсь об их устранении.

Иногда дедупликация не стоит затраченных усилий. Сравните первый пример кода в этом разделе с последним. Хотя код с дубликатами длиннее, он проще и прямолинейнее. Пример без дубликатов делает то же самое, но с добавлением цикла, новой переменной цикла `timeOfDay` и новой функции с параметром, которому также присвоено имя `timeOfDay`.

Дублирование кода способно вызывать ошибки, потому что оно усложняет целостное изменение кода. Если программа содержит несколько одинаковых фрагментов, стоит поместить код в функцию или цикл, чтобы он вызывался только один раз.

«Магические» числа

Не приходится удивляться тому, что в программировании используются числа. Но некоторые числа, встречающиеся в исходном коде, могут сбить с толку других программистов (или вас через пару недель после написания программы). Для примера возьмем число `604800` в следующей строке:

```
expiration = time.time() + 604800
```

Функция `time.time()` возвращает целое число, представляющее текущее время. Можно предположить, что переменная `expiration` представляет некий будущий момент, который наступит через 604 800 секунд. Но число `604800` выглядит загадочно: что оно означает? Комментарий поможет прояснить ситуацию:

```
expiration = time.time() + 604800  # Срок действия истекает через неделю.
```

Это хорошее решение, но еще лучше заменить такие «магические» числа константами. *Константы* представляют собой переменные, имена которых записаны в верхнем регистре, это означает, что они не должны изменяться после исходного присваивания. Обычно константы определяются как глобальные переменные в начале файла с исходным кодом:

```
# Константы для разных промежутков времени:
SECONDS_PER_MINUTE = 60
SECONDS_PER_HOUR   = 60 * SECONDS_PER_MINUTE
SECONDS_PER_DAY    = 24 * SECONDS_PER_HOUR
SECONDS_PER_WEEK   = 7  * SECONDS_PER_DAY
...
expiration = time.time() + SECONDS_PER_WEEK  # Срок действия истекает
                                             # через неделю.
```

Используйте отдельные константы для «магических» чисел, предназначенных для разных целей, даже если их числовые значения совпадают. Например, в колоде 52 карты, а в году 52 недели. Но если в вашей программе используются обе величины, нужно поступить примерно так:

```
NUM_CARDS_IN_DECK = 52
NUM_WEEKS_IN_YEAR = 52

print('This deck contains', NUM_CARDS_IN_DECK, 'cards.')
print('The 2-year contract lasts for', 2 * NUM_WEEKS_IN_YEAR, 'weeks.')
```

При выполнении этого кода результат будет выглядеть так:

```
This deck contains 52 cards.
The 2-year contract lasts for 104 weeks.

(Колода содержит 52 карты.
Двухлетний контракт длится 104 недели.)
```

Использование разных констант позволяет независимо изменять их в будущем. Конечно, значения констант не должны изменяться во время выполнения кода. Но это не означает, что программист никогда не обновит их в исходном коде. Например, если в будущей версии кода появится дополнительная карта-джокер, константу `cards` можно изменить независимо от `weeks`:

```
NUM_CARDS_IN_DECK = 53
NUM_WEEKS_IN_YEAR = 52
```

«Магическими» *числами* также иногда называют нечисловые значения. Например, строковые значения могут использоваться как константы. Возьмем следующую программу, которая предлагает пользователю указать направление и выводит

предупреждение, если пользователь выбрал `'north'`. Из-за опечатки `'nrth'` возникает ошибка, вследствие чего предупреждение не выводится:

```
while True:
    print('Set solar panel direction:')
    direction = input().lower()
    if direction in ('north', 'south', 'east', 'west'):
        break
print('Solar panel heading set to:', direction)
if direction == 'nrth':    ❶
    print('Warning: Facing north is inefficient for this panel.')
```

Найти такую ошибку нелегко: хотя в строке `'nrth'` совершена опечатка, она остается синтаксически правильным кодом Python. Программа не завершается аварийно, а предупреждение легко упустить из виду. Но если вы допустите ту же опечатку при использовании констант, ошибка будет обнаружена, потому что Python заметит, что константа `NRTH` не существует:

```
# Константы для разных промежутков времени:
NORTH = 'north'
SOUTH = 'south'
EAST = 'east'
WEST = 'west'

while True:
    print('Set solar panel direction:')
    direction = input().lower()
    if direction in (NORTH, SOUTH, EAST, WEST):
        break

print('Solar panel heading set to:', direction)
if direction == NRTH:      ❶
    print('Warning: Facing north is inefficient for this panel.')
```

Из-за исключения `NameError`, выдаваемого в строке кода с опечаткой `NRTH` ❶, ошибка становится очевидной при запуске программы:

```
Set solar panel direction:
west
Solar panel heading set to: west
Traceback (most recent call last):
  File "panelset.py", line 14, in <module>
    if direction == NRTH:
NameError: name 'NRTH' is not defined
```

«Магические» числа свидетельствуют о наличии запаха кода, потому что они не выполняют свое предназначение, затрудняют чтение и обновление кода и повышают риск опечаток, которые так трудно обнаружить. Проблема решается использованием констант.

Закомментированный и мертвый код

Поместить код в комментарий, чтобы он не выполнялся, — временная мера. Возможно, вы хотите пропустить часть строк, чтобы протестировать другую функциональность; закомментированные строки вы легко вернете позднее. Но если закомментированный код так и останется на месте, для читателя останется абсолютной тайной, почему он был удален и при каких условиях он может опять стать частью программы. Рассмотрим следующий пример:

```
doSomething()
#doAnotherThing()
doSomeImportantTask()
doAnotherThing()
```

Возникает целый ряд вопросов: почему вызов `doAnotherThing()` был закомментирован? Будет ли он снова включен в программу? Почему не был закомментирован второй вызов `doAnotherThing()`? Изначально в коде было два вызова `doAnotherThing()` или только один вызов, который переместился в точку после вызова `doSomeImportantTask()`? Почему закомментированный код не был удален, для этого есть какая-то причина? На все эти вопросы нет очевидных ответов.

Мертвым называется код, который недоступен или никогда не может быть выполнен на логическом уровне. Код внутри функции после команды `return`, команды `if`, условие которой всегда равно `False`, или код функции, которая никогда не вызывается в программе, — все это примеры мертвого кода. Чтобы увидеть пример на практике, введите следующую команду в интерактивной оболочке:

```
>>> import random
>>> def coinFlip():
...     if random.randint(0, 1):
...         return 'Heads!'
...     else:
...         return 'Tails!'
...     return 'The coin landed on its edge!'
...
>>> print(coinFlip())
Tails![1]
```

Строка `'The coin landed on its edge!'` является мертвым кодом, потому что код в блоках `if` и `else` возвращает управление до того, как программа сможет достичь этой строки. Мертвый код дезинформирует, поскольку читающие его программисты предполагают, что он составляет активную часть программы, тогда как по сути это закомментированный код.

---

[1] Heads — орел; Tails — решка; The coin landed on its edge — Монета встала ребром (*англ.*). — *Примеч. пер.*

Заглушки (stubs) являются исключением из этих правил. Они представляют собой заменители для будущего кода (например, функции и классы, которые еще не были реализованы). Вместо реального кода заглушка содержит команду `pass`, которая ничего не делает. (Она также называется *пустой операцией*.) Команда `pass` существует как раз для того, чтобы вы могли создавать заглушки в тех местах, где с точки зрения синтаксиса языка должен присутствовать какой-то код:

```
>>> def exampleFunction():
...     pass
...
```

Если вызвать эту функцию, она не сделает ничего. Вместо этого она лишь показывает, что когда-нибудь в нее будет добавлен код.

Также возможен другой вариант: чтобы предотвратить случайный вызов нереализованной функции, можно использовать заглушку из команды `raise NotImplementedError`. Тем самым вы покажете, что функция еще не готова к вызову:

```
>>> def exampleFunction():
...     raise NotImplementedError
...
>>> exampleFunction()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in exampleFunction
NotImplementedError
```

Исключение `NotImplementedError` предупредит вас о том, что в программе была случайно вызвана заглушка (функция или метод).

Закомментированный код и мертвый код считаются запахами кода, потому что они могут создать у программиста ошибочное впечатление, будто код является исполняемой частью программы. Вместо этого следует удалить их и использовать систему контроля версий (например, Git или Subversion) для отслеживания изменений. Контролю версий я посвятил главу 12. С системой контроля версий вы можете удалить код из своей программы, а при необходимости позднее вернуть его обратно.

Отладочный вывод

Отладочный вывод — это практика включения в программу временных вызовов `print()` для вывода значений переменных и повторного запуска программы.

Последовательность процесса, как правило, такова.

1. Обнаружение ошибки в программе.
2. Включение вызовов `print()` для некоторых переменных, чтобы узнать их текущие значения.