Code smells and best practices for clean code

Joachim Vandekerckhove

Introduction

Good coding practice is writing code that is not only effective, but also easy to read, adjust, and maintain.

Introduction

Good coding practice is writing code that is not only effective, but also easy to read, adjust, and maintain.

It is more like writing than it is like problem-solving: It requires some psychological insight, and thinking about the competence and working memory capacity of the reader.

Introduction

Good coding practice is writing code that is not only effective, but also easy to read, adjust, and maintain.

It is more like writing than it is like problem-solving: It requires some psychological insight, and thinking about the competence and working memory capacity of the reader.

More often than not, good coding practice is defined in terms of what not to do.

Introduction

Good coding practice is writing code that is not only effective, but also easy to read, adjust, and maintain.

It is more like writing than it is like problem-solving: It requires some psychological insight, and thinking about the competence and working memory capacity of the reader.

More often than not, good coding practice is defined in terms of what not to do.

With experience, you gain instincts that tell you when a piece of code is poorly written, hard to understand, or prone to generate bugs or runtime errors.

Introduction

Good coding practice is writing code that is not only effective, but also easy to read, adjust, and maintain.

It is more like writing than it is like problem-solving: It requires some psychological insight, and thinking about the competence and working memory capacity of the reader.

More often than not, good coding practice is defined in terms of what not to do.

With experience, you gain instincts that tell you when a piece of code is poorly written, hard to understand, or prone to generate bugs or runtime errors.

With these instincts, you will be able to tell when code smells bad.

Code smells

Code smells

Definition

Code smells are subjective indicators of potential problems in code that can impact the quality and maintainability of software.

Code smells

Definition

Code smells are subjective indicators of potential problems in code that can impact the quality and maintainability of software.

- Whether something is a code smell is a judgment about the quality of code, identifying issues such as code duplication, long method lengths, complexity, and lack of encapsulation.

Code smells

Definition

Code smells are subjective indicators of potential problems in code that can impact the quality and maintainability of software.

- Whether something is a code smell is a judgment about the quality of code, identifying issues such as code duplication, long method lengths, complexity, and lack of encapsulation.
- Code smells are not bugs or errors, but they make code more difficult to understand, maintain, and extend, and error-prone.

Code smells

Definition

Code smells are subjective indicators of potential problems in code that can impact the quality and maintainability of software.

- Whether something is a code smell is a judgment about the quality of code, identifying issues such as code duplication, long method lengths, complexity, and lack of encapsulation.
- Code smells are not bugs or errors, but they make code more difficult to understand, maintain, and extend, and error-prone.
- Code smells can lead to bugs, make code more difficult to maintain, and increase the effort required to fix problems.

Code smells

Definition

Code smells are subjective indicators of potential problems in code that can impact the quality and maintainability of software.

- Whether something is a code smell is a judgment about the quality of code, identifying issues such as code duplication, long method lengths, complexity, and lack of encapsulation.
- Code smells are not bugs or errors, but they make code more difficult to understand, maintain, and extend, and error-prone.
- Code smells can lead to bugs, make code more difficult to maintain, and increase the effort required to fix problems.

Code smells

Definition

Code smells are subjective indicators of potential problems in code that can impact the quality and maintainability of software.

- Whether something is a code smell is a judgment about the quality of code, identifying issues such as code duplication, long method lengths, complexity, and lack of encapsulation.
- Code smells are not bugs or errors, but they make code more difficult to understand, maintain, and extend, and error-prone.
- Code smells can lead to bugs, make code more difficult to maintain, and increase the effort required to fix problems.

Code smells are heuristics that indicate when to refactor.

Generic code smells

There are many code smells that are so common they've been given names.

Generic code smells

There are many code smells that are so common they've been given names.

The general idea is that a feeling of darkness and despair comes over you when you see some code and imagine having to do maintenance on it in the future:

Generic code smells

There are many code smells that are so common they've been given names.

The general idea is that a feeling of darkness and despair comes over you when you see some code and imagine having to do maintenance on it in the future:

- This is hard to read.

Generic code smells

There are many code smells that are so common they've been given names.

The general idea is that a feeling of darkness and despair comes over you when you see some code and imagine having to do maintenance on it in the future:

- This is hard to read.
- This is hard to test.

Generic code smells

There are many code smells that are so common they've been given names.

The general idea is that a feeling of darkness and despair comes over you when you see some code and imagine having to do maintenance on it in the future:

- This is hard to read.
- This is hard to test.
- If I change this, things elsewhere break.

Generic code smells

There are many code smells that are so common they've been given names.

The general idea is that a feeling of darkness and despair comes over you when you see some code and imagine having to do maintenance on it in the future:

- This is hard to read.
- This is hard to test.
- If I change this, things elsewhere break.
- I find myself changing this back and forth.

Application-level smells

Mysterious name

- Functions, modules, variables or classes that are named in a way that does not communicate what they do or how to use them.

```
doTheThing(a,aa,flag)
```

Mysterious name

− Functions, modules, variables or classes that are named in a way that does not communicate what they do or how to use them.

```
doTheThing(a,aa,flag)
```

+ Use meaningful and descriptive names.

```
marginalLikelihood(binomialData, prior, useCache)
```

Mysterious name

– Functions, modules, variables or classes that are named in a way that does not communicate what they do or how to use them.

```
doTheThing(a,aa,flag)
```

+ Use meaningful and descriptive names.

```
marginalLikelihood(binomialData, prior, useCache)
```

+ Use naming conventions and abbreviations that are widely recognized and understood.

```
margLik(binomialData, prior, useCache)
```

Duplicated code

- Identical or very similar code exists in more than one location.

```
np.mean(samples[0][’alpha’].flatten())
np.mean(samples[0][’gamma’].flatten())
np.mean(samples[0][’delta’].flatten())
```

Duplicated code

– Identical or very similar code exists in more than one location.

```
np.mean(samples[0]['alpha'].flatten())
np.mean(samples[0]['gamma'].flatten())
np.mean(samples[0]['delta'].flatten())
```

+ Extract duplicated code into a reusable function or class.

```
def chainMean(samples, parameter):
    chain = samples[0][parameter].flatten()
    return numpy.mean(chain)
```

```
chainMean(samples, 'alpha')
chainMean(samples, 'gamma')
chainMean(samples, 'delta')
```

Duplicated code

– Identical or very similar code exists in more than one location.

```
np.mean(samples[0][’alpha’].flatten())
np.mean(samples[0][’gamma’].flatten())
np.mean(samples[0][’delta’].flatten())
```

+ Extract duplicated code into a reusable function or class.

```
def chainMean(samples, parameter):
    chain = samples[0][parameter].flatten()
    return numpy.mean(chain)
```

```
chainMean(samples, ’alpha’)
chainMean(samples, ’gamma’)
chainMean(samples, ’delta’)
```

+ Ensure that the extracted code is well-tested, so that it can be easily maintained and updated.

```
def testChainMean(unittest.TestCase):
    pass
```

Shotgun surgery

– Single changes often need to be applied to multiple classes or methods at the same time.

```
def integrand1(self, p):
    return self.pdf(p, self.n, self.k) * self.prior1(p)
def integrand2(self, p):
    return self.pdf(p, self.n, self.k) * self.prior2(p)
```

Shotgun surgery

– Single changes often need to be applied to multiple classes or methods at the same time.

```
def integrand1(self, p):
    return self.pdf(p, self.n, self.k) * self.prior1(p)
def integrand2(self, p):
    return self.pdf(p, self.n, self.k) * self.prior2(p)
```

+ Identify the commonality between the classes that need to be modified and extract it into a common class or function.

```
def integrand(self, prior, p):
    return self.pdf(p, self.n, self.k) * prior(p)
```

Shotgun surgery

– Single changes often need to be applied to multiple classes or methods at the same time.

```
def integrand1(self, p):
    return self.pdf(p, self.n, self.k) * self.prior1(p)
def integrand2(self, p):
    return self.pdf(p, self.n, self.k) * self.prior2(p)
```

+ Identify the commonality between the classes that need to be modified and extract it into a common class or function.

```
def integrand(self, prior, p):
    return self.pdf(p, self.n, self.k) * prior(p)
```

+ Ensure that the extracted code is well-tested, so that it can be easily maintained and updated.

Shotgun surgery

− Single changes often need to be applied to multiple classes or methods at the same time.

```
def integrand1(self, p):
    return self.pdf(p, self.n, self.k) * self.prior1(p)
def integrand2(self, p):
    return self.pdf(p, self.n, self.k) * self.prior2(p)
```

+ Identify the commonality between the classes that need to be modified and extract it into a common class or function.

```
def integrand(self, prior, p):
    return self.pdf(p, self.n, self.k) * prior(p)
```

+ Ensure that the extracted code is well-tested, so that it can be easily maintained and updated.

+ Don’t split a single responsibility among classes or methods.

Variable mutations

– Frequently changing what a particular identifier refers to.

```matlab
function p = binomial_pdf(obj, p)
    p = nchoosek(obj.n, obj.k) .* (p.^obj.k) .* ...
        ((1 - p).^(obj.n - obj.k));
end
```

Variable mutations

– Frequently changing what a particular identifier refers to.

```
function p = binomial_pdf(obj, p)
    p = nchoosek(obj.n, obj.k) .* (p.^obj.k) .* ...
        ((1 - p).^(obj.n - obj.k));
end
```

+ Refactor the code to avoid changing the reference of a variable, instead create a new variable with a new reference.

```
function outputArg = binomial_pdf(obj, prob)
    outputArg = nchoosek(obj.n, obj.k) .* ...
        (prob.^obj.k) .* ((1 - prob).^(obj.n - obj.k));
end
```

Variable mutations

− Frequently changing what a particular identifier refers to.

```
function p = binomial_pdf(obj, p)
    p = nchoosek(obj.n, obj.k) .* (p.^obj.k) .* ...
        ((1 - p).^(obj.n - obj.k));
end
```

+ Refactor the code to avoid changing the reference of a variable, instead create a new variable with a new reference.

```
function outputArg = binomial_pdf(obj, prob)
    outputArg = nchoosek(obj.n, obj.k) .* ...
        (prob.^obj.k) .* ((1 - prob).^(obj.n - obj.k));
end
```

+ Consider using constants or read-only variables where appropriate to reduce the likelihood of unexpected mutations.

Uncontrolled side effects

- Methods modify variables beyond scope (like globals or complex variables passed by reference).

Respect scope.

Uncontrolled side effects

− Methods modify variables beyond scope (like globals or complex variables passed by reference).

+ Refactor the code to avoid modifying variables outside of the method scope.

Respect scope.

Uncontrolled side effects

- – Methods modify variables beyond scope (like globals or complex variables passed by reference).
- + Refactor the code to avoid modifying variables outside of the method scope.
- + Always use functional programming techniques, such as passing data through function arguments and return values, to reduce the likelihood of uncontrolled side effects.

Respect scope.

Uncontrolled side effects | Python passes objects by reference

```python
class myClass:
    def __init__(self, X):
        self.X = X

def getX(obj):
    X = obj.X
    obj.X = None
    return X

n = myClass(5)

(n.X, getX(n), n.X)
```

```
(5, 5, None)
```

Uncontrolled side effects | MATLAB passes objects by value

```
classdef myClass
    properties
        X
    end
    methods
        function obj = myClass(X)
            obj.X = X;
        end
        function X = getX(obj)
            X = obj.X;
            obj.X = [];
        end
    end
end
```

```
>> n = myClass(5);
>> disp([n.X, n.getX(), n.X])
     5     5     5
```

Other application-level smells

- Data clumps

Other application-level smells

- Data clumps
  - Multiple related variables that are often passed together, indicating a need for a class.

Other application-level smells

- Data clumps
  - – Multiple related variables that are often passed together, indicating a need for a class.
  - + Refactor the code to extract the related variables into a class.

Other application-level smells

- Data clumps
  - – Multiple related variables that are often passed together, indicating a need for a class.
  - + Refactor the code to extract the related variables into a class.
  - + Easier to understand and maintain.

Other application-level smells

- Data clumps
  - – Multiple related variables that are often passed together, indicating a need for a class.
  - + Refactor the code to extract the related variables into a class.
  - + Easier to understand and maintain.
- Speculative generality

Other application-level smells

- Data clumps
  - Multiple related variables that are often passed together, indicating a need for a class.
  - + Refactor the code to extract the related variables into a class.
  - + Easier to understand and maintain.
- Speculative generality
  - Writing code for functionality that may not be needed in the future adds unnecessary complexity and makes the code difficult to maintain.

Other application-level smells

- Data clumps
  - Multiple related variables that are often passed together, indicating a need for a class.
  - + Refactor the code to extract the related variables into a class.
  - + Easier to understand and maintain.
- Speculative generality
  - Writing code for functionality that may not be needed in the future adds unnecessary complexity and makes the code difficult to maintain.
- High cyclomatic complexity

Other application-level smells

- Data clumps
  - Multiple related variables that are often passed together, indicating a need for a class.
  - + Refactor the code to extract the related variables into a class.
  - + Easier to understand and maintain.
- Speculative generality
  - Writing code for functionality that may not be needed in the future adds unnecessary complexity and makes the code difficult to maintain.
- High cyclomatic complexity
  - Every possible path through a function adds complexity.

Other application-level smells

- Data clumps
    - Multiple related variables that are often passed together, indicating a need for a class.
    + Refactor the code to extract the related variables into a class.
    + Easier to understand and maintain.
- Speculative generality
    - Writing code for functionality that may not be needed in the future adds unnecessary complexity and makes the code difficult to maintain.
- High cyclomatic complexity
    - Every possible path through a function adds complexity.
    + It may be possible to simplify the logic, or this needs to be multiple functions.

Method-level smells

Too many parameters

- Makes calling and testing a function complicated

```
function p = plot_experiment(ivData, ivNames, ...
    dvData, dvNames, covariates, covariateNames, ...
    experimentName, aesthetic)
    ... do stuff ...
end
```

Too many parameters

- Makes calling and testing a function complicated

```
function p = plot_experiment(ivData, ivNames, ...
    dvData, dvNames, covariates, covariateNames, ...
    experimentName, aesthetic)
    ... do stuff ...
end
```

- May indicate that the purpose of the function is ill-conceived

Too many parameters

– Makes calling and testing a function complicated

```
function p = plot_experiment(ivData, ivNames, ...
    dvData, dvNames, covariates, covariateNames, ...
    experimentName, aesthetic)
    ... do stuff ...
end
```

– May indicate that the purpose of the function is ill-conceived

+ Refactor so responsibility is assigned in a more clean-cut way

```
classdef experiment
    ... Class to contain experiment data ...
```

Long method

- A method that has grown too large

Long method

- A method that has grown too large
- Difficult to understand and maintain

Long method

− A method that has grown too large

− Difficult to understand and maintain

+ Refactor into smaller, more focused methods that each perform a single, specific task

Excessively long identifiers

- Naming conventions used to provide disambiguation that should be implicit in the software architecture

```
compute_marginal_likelihood_from_binomial_data_and_priors(\
    binomialData, prior, useCache)
```

Excessively long identifiers

- Naming conventions used to provide disambiguation that should be implicit in the software architecture

```
compute_marginal_likelihood_from_binomial_data_and_priors(\
    binomialData, prior, useCache)
```

- Can make code harder to read and understand

Excessively long identifiers

– Naming conventions used to provide disambiguation that should be implicit in the software architecture

```
compute_marginal_likelihood_from_binomial_data_and_priors(\
    binomialData, prior, useCache)
```

– Can make code harder to read and understand

+ Use concise but descriptive names

```
marginalLikelihood(binomialData, prior, useCache)
```

Excessively short identifiers

– Short, non-descriptive names make code hard to read and understand

```
ml(d1 , pr , uc)
```

Excessively short identifiers

– Short, non-descriptive names make code hard to read and understand

```
ml(d1, pr, uc)
```

+ Use descriptive but concise names

```
marginalLikelihood(binomialData, prior, useCache)
```

Excessively long line of code

- Makes code difficult to read, understand, debug, refactor, or identify possibilities for software reuse

```
subtab = data.loc[subset].groupby(factors)[value \
    ].agg([numpy.mean, numpy.std, len])
```

Excessively long line of code

– Makes code difficult to read, understand, debug, refactor, or identify possibilities for software reuse

```
subtab = data.loc[subset].groupby(factors)[value \
    ].agg([numpy.mean, numpy.std, len])
```

+ Break up long lines into smaller, more manageable chunks

```
statistics_list  = [numpy.mean, numpy.std, len]
subset           = data.loc[subset]
grouped_subset   = subset.groupby(factors)
grouped_value    = grouped_subset[value]
aggregated_value = grouped_value.agg(statistics_list)
```

Excessive comments

- A comment on an attribute setter/getter is a good example

Excessive comments

- A comment on an attribute setter/getter is a good example
- “Code deodorant”

Excessive comments

- A comment on an attribute setter/getter is a good example
- “Code deodorant”
- Can make code harder to read and maintain because a change in code leads to a change in comment

Excessive comments

– A comment on an attribute setter/getter is a good example

– “Code deodorant”

– Can make code harder to read and maintain because a change in code leads to a change in comment

+ Write short methods with descriptive titles so they don’t need much explanation

Excessive comments

- – A comment on an attribute setter/getter is a good example
- – “Code deodorant”
- – Can make code harder to read and maintain because a change in code leads to a change in comment
- + Write short methods with descriptive titles so they don’t need much explanation
- + Practice contractual programming: document the interface carefully

Class-level smells

Class-level smells

- **Large class**: a class that contains many unrelated methods.

Class-level smells

- **Large class**: a class that contains many unrelated methods.
- **Lazy class**: a class that does too little.

Class-level smells

- **Large class**: a class that contains many unrelated methods.
- **Lazy class**: a class that does too little.
- **Feature envy**: a class that uses methods of another class excessively.

Class-level smells

- **Large class**: a class that contains many unrelated methods.
- **Lazy class**: a class that does too little.
- **Feature envy**: a class that uses methods of another class excessively.
- **Excessive use of literals**: these should be coded as named constants, to improve readability and to avoid programming errors.

Best practices for clean code

Naming

- Use clear and descriptive names for variables, functions, classes, and other identifiers.

Naming

- Use clear and descriptive names for variables, functions, classes, and other identifiers.
- The names should clearly indicate the purpose and behavior of the code.

Naming

- Use clear and descriptive names for variables, functions, classes, and other identifiers.
- The names should clearly indicate the purpose and behavior of the code.
- Example:

Naming

- Use clear and descriptive names for variables, functions, classes, and other identifiers.
- The names should clearly indicate the purpose and behavior of the code.
- Example:
    - Instead of using a variable name like `x`, use `customerName`

Naming

- Use clear and descriptive names for variables, functions, classes, and other identifiers.
- The names should clearly indicate the purpose and behavior of the code.
- Example:
    - Instead of using a variable name like `x`, use `customerName`
    - Instead of using a function name like `process`, use `calculateDiscount`

Naming Conventions

- Bad naming practices can make code difficult to read and understand

Naming Conventions

- Bad naming practices can make code difficult to read and understand
- Good naming practices make code more self-documenting and easier to maintain

Naming Conventions

- Bad naming practices can make code difficult to read and understand
- Good naming practices make code more self-documenting and easier to maintain

Naming Conventions

- Bad naming practices can make code difficult to read and understand
- Good naming practices make code more self-documenting and easier to maintain

```python
# Bad naming practice
lst = [1, 2, 3]
dct = {"a": 1, "b": 2}
def fnc(a, b):
    return a + b

class Cls:
    def __init__(self):
        self.x = None
        self.y = None
    def mtd(self):
        pass
```

Naming Conventions

- Bad naming practices can make code difficult to read and understand
- Good naming practices make code more self-documenting and easier to maintain

```python
# Bad naming practice
lst = [1, 2, 3]
dct = {"a": 1, "b": 2}
def fnc(a, b):
    return a + b

class Cls:
    def __init__(self):
        self.x = None
        self.y = None
    def mtd(self):
        pass
```

```python
# Good naming practice
numbers_list = [1, 2, 3]
data_dict = {"height": 1, "weight": 2}
def addNumbers(firstNumber, secondNumber):
    return firstNumber + secondNumber

class DataProcessor:
    def __init__(self):
        self.current_data = None
        self.previous_data = None
    def process_data(self):
        pass
```

Functions

- Keep functions small and focused.

Functions

- Keep functions small and focused.
- A function should do one thing and do it well.

Functions

- Keep functions small and focused.
- A function should do one thing and do it well.
- Avoid long functions with multiple responsibilities.

Functions

- Keep functions small and focused.
- A function should do one thing and do it well.
- Avoid long functions with multiple responsibilities.
- Example:

Functions

- Keep functions small and focused.
- A function should do one thing and do it well.
- Avoid long functions with multiple responsibilities.
- Example:
    - Instead of having a long function that retrieves data from a database, processes it, and sends an email all in one, have three separate functions: one for retrieving data, one for processing data, and one for sending emails.

Refactor functions

```python
def calculate_and_save_results(numbers):
    result = 0
    for number in numbers:
        result += number
    with open("result.txt", "w") as f:
        f.write(str(result))
    return result
```

```python
def calculate_sum(numbers):
    result = 0
    for number in numbers:
        result += number
    return result

def save_to_file(result, file_path):
    with open(file_path, "w") as f:
        f.write(str(result))

numbers = [1, 2, 3, 4]
result = calculate_sum(numbers)
save_to_file(result, "result.txt")
```

Comments

- Use comments only when they provide additional information that is not already clear from the code.

Comments

- Use comments only when they provide additional information that is not already clear from the code.
- Avoid writing comments that simply repeat the code.

Comments

- Use comments only when they provide additional information that is not already clear from the code.
- Avoid writing comments that simply repeat the code.
- Example:

Comments

- Use comments only when they provide additional information that is not already clear from the code.
- Avoid writing comments that simply repeat the code.
- Example:
    - Instead of writing a comment like "increment the value of x" before "x += 1"... don't write that

Comments

- Use comments only when they provide additional information that is not already clear from the code.
- Avoid writing comments that simply repeat the code.
- Example:
    - Instead of writing a comment like "increment the value of x" before "x += 1"... don't write that
- If you find you need to explain your code a lot, you might want to simplify the code rather than apologizing for it

Bad Commenting Practice

```python
def calculate_sum(numbers):
    # Calculating the sum of numbers
    result = 0
    for number in numbers:
        result += number
    # Return result
    return result
```

Best Commenting Practice

```python
def calculate_sum(numbers):
    """
    Calculate the sum of a list of numbers.

    Arguments:
    numbers -- list of numbers to be summed

    Returns:
    result -- the sum of the numbers
    """
    result = 0
    for number in numbers:
        result += number
    return result
```

Formatting

- Use consistent formatting throughout the codebase.

Formatting

- Use consistent formatting throughout the codebase.
- This makes the code easier to read and understand.

Formatting

- Use consistent formatting throughout the codebase.
- This makes the code easier to read and understand.
- Example:

Formatting

- Use consistent formatting throughout the codebase.
- This makes the code easier to read and understand.
- Example:
    - Use a consistent indentation level throughout the code.

Formatting

- Use consistent formatting throughout the codebase.
- This makes the code easier to read and understand.
- Example:
    - Use a consistent indentation level throughout the code.
    - Use consistent capitalization for variables and functions.

Formatting

- Use consistent formatting throughout the codebase.
- This makes the code easier to read and understand.
- Example:
    - Use a consistent indentation level throughout the code.
    - Use consistent capitalization for variables and functions.
    - Use consistent spacing and alignment.

Bad Formatting Practice

```matlab
function outputArg = calculate_sum(numbers)
outputArg = 0;
for i = 1:length(numbers)
outputArg=outputArg+numbers(i);
end
end
```

```matlab
function outputArg = calculate_sum(numbers)
    outputArg = 0;
    for i = 1:length(numbers)
        outputArg = outputArg + numbers(i);
    end
end
```

Error handling

- Handle errors/exceptions in a consistent and predictable way.

Error handling

- Handle errors/exceptions in a consistent and predictable way.
- Use exceptions to indicate what has gone wrong.

Error handling

- Handle errors/exceptions in a consistent and predictable way.
- Use exceptions to indicate what has gone wrong.
- In general, do not try to fix a user's input error. Wrong input should give an informative error message to set the user straight. It should never quietly give a result that might not be what the user wanted.

Error Handling

Inferior Error Handling Practice

```
def centered_normpdf(x, s):
    return (1 / (math.sqrt(2 * math.pi) * s)) * \
        math.exp(-0.5 * (x / s) ** 2)

print(centered_normpdf(0, 0))
# ZeroDivisionError: float division by zero
```

Better Error Handling Practice

```
def centered_normpdf(x, sigma):
    if sigma == 0:
        raise ValueError("The standard deviation" +
            " must be non-zero.")
    return (1 / (math.sqrt(2 * math.pi) * sigma)) * \
        math.exp(-0.5 * (x / sigma) ** 2)

print(centered_normpdf(0, 0))
# ValueError: The standard deviation must be non-zero.
```

Refactoring

- Regularly review and refactor the code to improve its design and maintainability.

Refactoring

- Regularly review and refactor the code to improve its design and maintainability.
- Make small, incremental changes to the code rather than large, sweeping changes.

Refactoring

- Regularly review and refactor the code to improve its design and maintainability.
- Make small, incremental changes to the code rather than large, sweeping changes.
- Example:

Refactoring

- Regularly review and refactor the code to improve its design and maintainability.
- Make small, incremental changes to the code rather than large, sweeping changes.
- Example:
  - Instead of rewriting an entire module, extract the reusable parts into a separate function or class.

Refactoring

- Regularly review and refactor the code to improve its design and maintainability.
- Make small, incremental changes to the code rather than large, sweeping changes.
- Example:
  - Instead of rewriting an entire module, extract the reusable parts into a separate function or class.
  - Instead of adding new features to a class with many responsibilities, extract the new feature into a separate class.

Final good practices

- Unit testing

Final good practices

- Unit testing
  - Write unit tests to ensure that the code behaves as expected and to catch regressions.

Final good practices

- Unit testing
    - Write unit tests to ensure that the code behaves as expected and to catch regressions.
    - A good test suite gives you confidence to make changes to the code without fear of breaking existing functionality.

Final good practices

- Unit testing
  - Write unit tests to ensure that the code behaves as expected and to catch regressions.
  - A good test suite gives you confidence to make changes to the code without fear of breaking existing functionality.
- Continual learning

Final good practices

- Unit testing
  - Write unit tests to ensure that the code behaves as expected and to catch regressions.
  - A good test suite gives you confidence to make changes to the code without fear of breaking existing functionality.
- Continual learning
  - Stay up-to-date with new technologies and industry best practices.

Final good practices

- Unit testing
    - Write unit tests to ensure that the code behaves as expected and to catch regressions.
    - A good test suite gives you confidence to make changes to the code without fear of breaking existing functionality.
- Continual learning
    - Stay up-to-date with new technologies and industry best practices.
    - Continuously improve your skills and knowledge.

Final good practices

- Unit testing
    - Write unit tests to ensure that the code behaves as expected and to catch regressions.
    - A good test suite gives you confidence to make changes to the code without fear of breaking existing functionality.
- Continual learning
    - Stay up-to-date with new technologies and industry best practices.
    - Continuously improve your skills and knowledge.
    - Example:

Final good practices

- Unit testing
  - Write unit tests to ensure that the code behaves as expected and to catch regressions.
  - A good test suite gives you confidence to make changes to the code without fear of breaking existing functionality.
- Continual learning
  - Stay up-to-date with new technologies and industry best practices.
  - Continuously improve your skills and knowledge.
  - Example:
    - Read books and articles on software development.

Final good practices

- Unit testing
  - Write unit tests to ensure that the code behaves as expected and to catch regressions.
  - A good test suite gives you confidence to make changes to the code without fear of breaking existing functionality.
- Continual learning
  - Stay up-to-date with new technologies and industry best practices.
  - Continuously improve your skills and knowledge.
  - Example:
    - Read books and articles on software development.
    - Attend conferences, meetups, and other events.

Final good practices

- Unit testing
  - Write unit tests to ensure that the code behaves as expected and to catch regressions.
  - A good test suite gives you confidence to make changes to the code without fear of breaking existing functionality.
- Continual learning
  - Stay up-to-date with new technologies and industry best practices.
  - Continuously improve your skills and knowledge.
  - Example:
    - Read books and articles on software development.
    - Attend conferences, meetups, and other events.
    - Experiment with new technologies and programming languages.