Coding Standards

Coding Standards

- Naming conventions
  - *how to choose meaningful names for classes, methods, and variables*
  - *camelCase vs. snake_case vs. kebab-case vs. PascalCase for each of the above*
- Formatting
  - *spaces vs. tabs, max line length, ...*
  - *braces on same vs. next line, alignment of method parameters, etc.*
- Commenting and documentation
- Best practices

```
class OrderManager(private val server: Server) {

    // Place an order and return a corresponding OrderResult
    fun placeOrder(orderDetails: OrderDetails): OrderResult {
        val response = server.submitOrder(orderDetails)

        if (response.isSuccess) {
            return OrderResult.Success(response.orderId)
        }
        return OrderResult.Failure(response.errorMessage)
    }

    // Cancel an order by ID; returns true if successful, false otherwise
    fun cancelOrder(orderId: String): Boolean {
        val response = server.cancelOrder(orderId)

        return response.isSuccess
    }
}
```

```
class OrderManager(private val server: Server) {

    // Place an order and return a corresponding OrderResult
    fun placeOrder(orderDetails: OrderDetails): OrderResult {
        val response = server.submitOrder(orderDetails)

        if (response.isSuccess) {
            return OrderResult.Success(response.orderId)
        }
        return OrderResult.Failure(response.errorMessage)
    }

    // Cancel an order by ID; returns true if successful, false otherwise
    fun cancelOrder(orderId: String): Boolean {
        val response = server.cancelOrder(orderId)

        return response.isSuccess
    }
}
```

```
class OrderManager(private val server: Server) {

    // Place an order and return a corresponding OrderResult
    fun placeOrder(details: OrderDetails) = server.submitOrder(details).let {
        if (it.isSuccess) OrderResult.Success(it.orderId) else OrderResult.Failure(it.errorMessage)
    }

    // Cancel an order by ID; returns true if successful, false otherwise
    fun cancelOrder(orderId: String): Boolean =
        server.cancelOrder(orderId).isSuccess
}
```

```
class OrderManager(private val server: Server) {

    // Place an order and return a corresponding OrderResult
    fun placeOrder(orderDetails: OrderDetails): OrderResult {
        val response = server.submitOrder(orderDetails)

        if (response.isSuccess) {
            return OrderResult.Success(response.orderId)
        }
        return OrderResult.Failure(response.errorMessage)
    }

    // Cancel an order by ID; returns true if successful, false otherwise
    fun cancelOrder(orderId: String): Boolean {
        val response = server.cancelOrder(orderId)

        return response.isSuccess
    }
}
```

```
class OrderManager(private val server: Server) {

    // Place an order and return a corresponding OrderResult
    fun placeOrder(details: OrderDetails) = server.submitOrder(details).let {
        if (it.isSuccess) OrderResult.Success(it.orderId) else OrderResult.Failure(it.errorMessage)
    }

    // Cancel an order by ID; returns true if successful, false otherwise
    fun cancelOrder(orderId: String): Boolean =
        server.cancelOrder(orderId).isSuccess
}
```

```
/**
 * Manages food orders by interfacing with a server for order operations.
 *
 * @property server The server used for order operations.
 */
class OrderManager(private val server: Server) {

    /**
     * Places a new order with specified details.
     *
     * Returns an [OrderResult] indicating the outcome.
     *
     * @param orderDetails Details of the order being placed.
     * @return [OrderResult] indicating success or failure.
     */
    fun placeOrder(orderDetails: OrderDetails): OrderResult =
        server.submitOrder(orderDetails).let { response ->
            if (response.isSuccess) OrderResult.Success(response.orderId)
            else OrderResult.Failure(response.errorMessage)
        }

    /**
     * Cancels an order by its ID.
     *
     * Returns a boolean indicating the operation's success.
     *
     * @param orderId The ID of the order to cancel.
     * @return True if cancellation was successful, false otherwise.
     */
    fun cancelOrder(orderId: String): Boolean =
        server.cancelOrder(orderId).isSuccess
}
```

Linux kernel coding style

This is a short document describing the preferred coding style for the linux kernel. Coding style is very personal, and I won't **force** my views on anybody, but this is what goes for anything that I have to be able to maintain, and I'd prefer it for most other things too. Please at least consider the points made here.

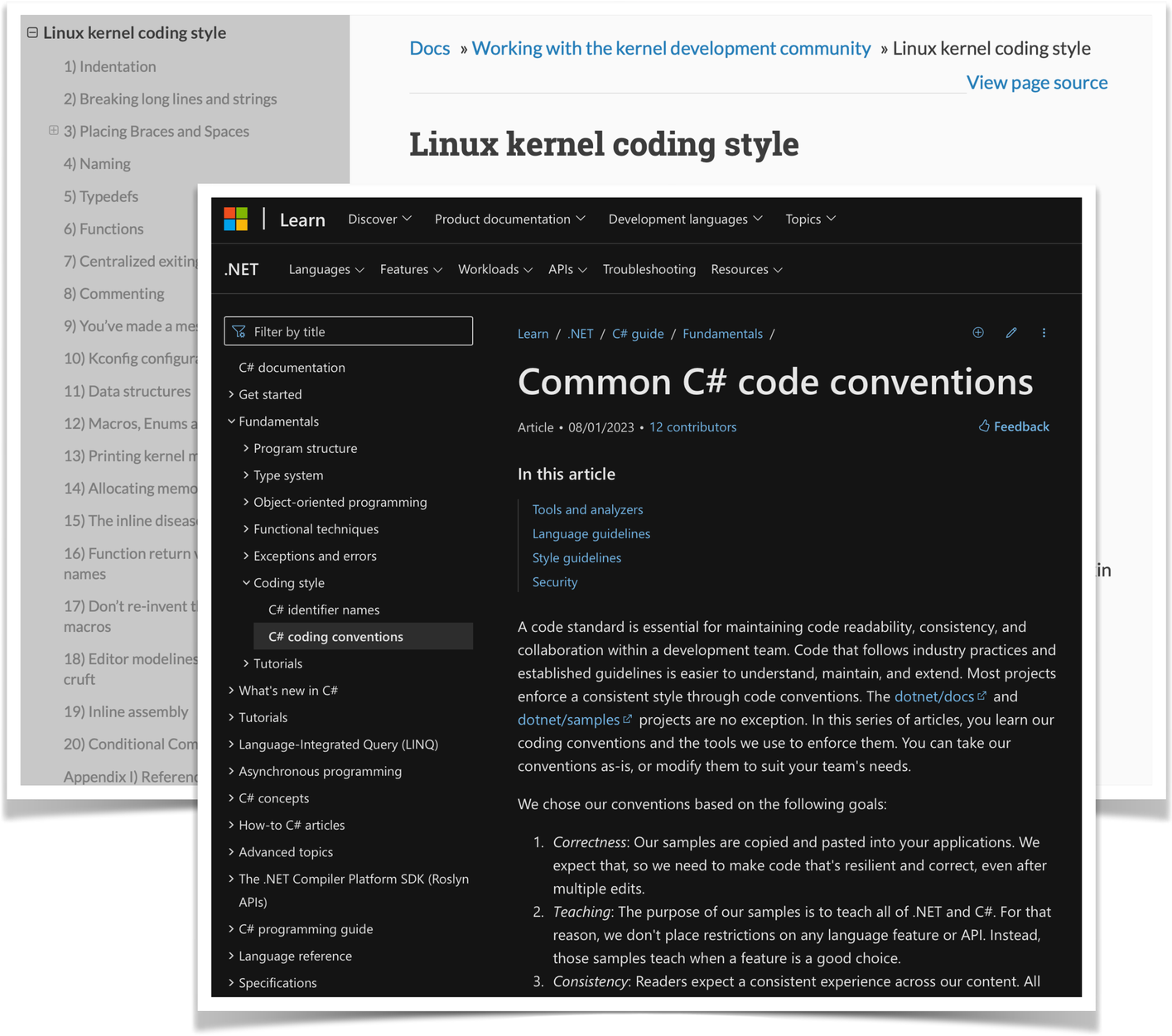
First off, I'd suggest printing out a copy of the GNU coding standards, and NOT read it. Burn them, it's a great symbolic gesture.

Anyway, here goes:

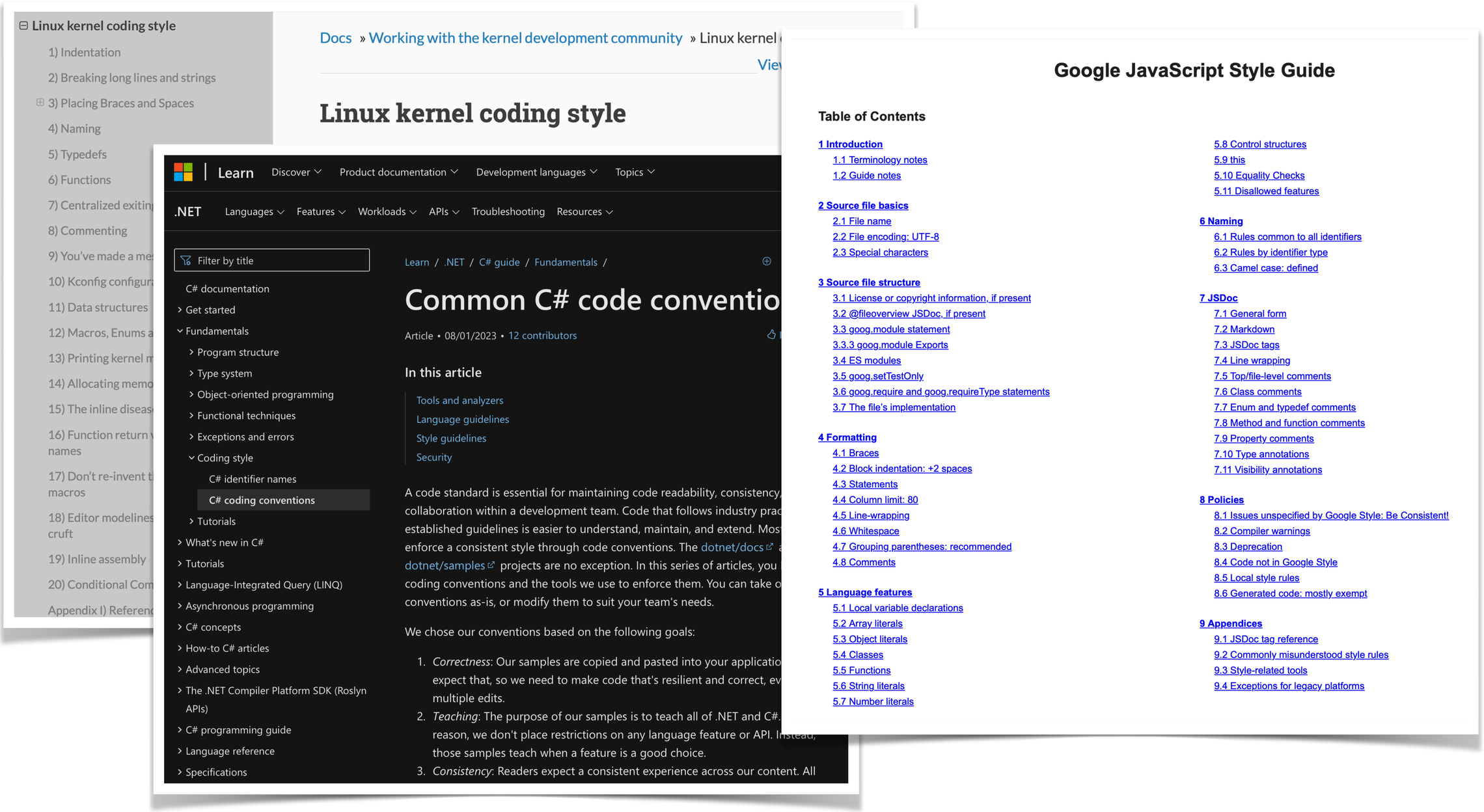
1) Indentation

Tabs are 8 characters, and thus indentations are also 8 characters. There are heretic movements that try to make indentations 4 (or even 2!) characters deep, and that is akin to trying to define the value of PI to be 3.

Rationale: The whole idea behind indentation is to clearly define where a block of control starts and ends. Especially when you've been looking at your screen for 20 straight hours, you'll find it a lot easier to see how the indentation works if you have large indentations.


Linux kernel coding style
1) Indentation
2) Breaking long lines and strings
3) Placing Braces and Spaces
4) Naming
5) Typedefs
6) Functions
7) Centralized exiting
8) Commenting
9) You've made a me
10) Kconfig configura
11) Data structures
12) Macros, Enums a
13) Printing kernel m
14) Allocating memo
15) The inline disease
16) Function return v
names
17) Don't re-invent t
macros
18) Editor modelines
cruft
19) Inline assembly
20) Conditional Com
Appendix I) Referenc
Docs » Working with the kernel development community » Linux kernel coding style
View page source
Linux kernel coding style
in

Learn
Discover
Product documentation
Development languages
Topics
.NET
Languages
Features
Workloads
APIs
Troubleshooting
Resources
Filter by title
C# documentation
Get started
Fundamentals
Program structure
Type system
Object-oriented programming
Functional techniques
Exceptions and errors
Coding style
C# identifier names
C# coding conventions
Tutorials
What's new in C#
Tutorials
Language-Integrated Query (LINQ)
Asynchronous programming
C# concepts
How-to C# articles
Advanced topics
The .NET Compiler Platform SDK (Roslyn APIs)
C# programming guide
Language reference
Specifications
Learn / .NET / C# guide / Fundamentals /
Common C# code conventions
Article • 08/01/2023 • 12 contributors
Feedback
In this article
Tools and analyzers
Language guidelines
Style guidelines
Security
A code standard is essential for maintaining code readability, consistency, and collaboration within a development team. Code that follows industry practices and established guidelines is easier to understand, maintain, and extend. Most projects enforce a consistent style through code conventions. The dotnet/docs and dotnet/samples projects are no exception. In this series of articles, you learn our coding conventions and the tools we use to enforce them. You can take our conventions as-is, or modify them to suit your team's needs.
We chose our conventions based on the following goals:
1. Correctness: Our samples are copied and pasted into your applications. We expect that, so we need to make code that's resilient and correct, even after multiple edits.
2. Teaching: The purpose of our samples is to teach all of .NET and C#. For that reason, we don't place restrictions on any language feature or API. Instead, those samples teach when a feature is a good choice.
3. Consistency: Readers expect a consistent experience across our content. All



Linux kernel coding style
1) Indentation
2) Breaking long lines and strings
3) Placing Braces and Spaces
4) Naming
5) Typedefs
6) Functions
7) Centralized exiting
8) Commenting
9) You've made a me
10) Kconfig configura
11) Data structures
12) Macros, Enums a
13) Printing kernel m
14) Allocating memo
15) The inline disease
16) Function return v names
17) Don't re-invent t macros
18) Editor modelines cruft
19) Inline assembly
20) Conditional Com
Appendix I) Referenc

Docs » Working with the kernel development community » Linux kernel
Vie

Linux kernel coding style

Learn Discover Product documentation Development languages Topics
.NET Languages Features Workloads APIs Troubleshooting Resources
Filter by title
C# documentation
Get started
Fundamentals
Program structure
Type system
Object-oriented programming
Functional techniques
Exceptions and errors
Coding style
C# identifier names
C# coding conventions
Tutorials
What's new in C#
Tutorials
Language-Integrated Query (LINQ)
Asynchronous programming
C# concepts
How-to C# articles
Advanced topics
The .NET Compiler Platform SDK (Roslyn APIs)
C# programming guide
Language reference
Specifications

Learn / .NET / C# guide / Fundamentals /

Common C# code conventio

Article • 08/01/2023 • 12 contributors

In this article

Tools and analyzers
Language guidelines
Style guidelines
Security

A code standard is essential for maintaining code readability, consistency, collaboration within a development team. Code that follows industry prac established guidelines is easier to understand, maintain, and extend. Mos enforce a consistent style through code conventions. The dotnet/docs dotnet/samples projects are no exception. In this series of articles, you coding conventions and the tools we use to enforce them. You can take o conventions as-is, or modify them to suit your team's needs.

We chose our conventions based on the following goals:

1. *Correctness*: Our samples are copied and pasted into your applicatio expect that, so we need to make code that's resilient and correct, ev multiple edits.
2. *Teaching*: The purpose of our samples is to teach all of .NET and C#. reason, we don't place restrictions on any language feature or API. Instead, those samples teach when a feature is a good choice.
3. *Consistency*: Readers expect a consistent experience across our content. All

Google JavaScript Style Guide

Table of Contents

1 Introduction
1.1 Terminology notes
1.2 Guide notes

2 Source file basics
2.1 File name
2.2 File encoding: UTF-8
2.3 Special characters

3 Source file structure
3.1 License or copyright information, if present
3.2 @fileoverview JSDoc, if present
3.3 goog.module statement
3.3.3 goog.module Exports
3.4 ES modules
3.5 goog.setTestOnly
3.6 goog.require and goog.requireType statements
3.7 The file's implementation

4 Formatting
4.1 Braces
4.2 Block indentation: +2 spaces
4.3 Statements
4.4 Column limit: 80
4.5 Line-wrapping
4.6 Whitespace
4.7 Grouping parentheses: recommended
4.8 Comments

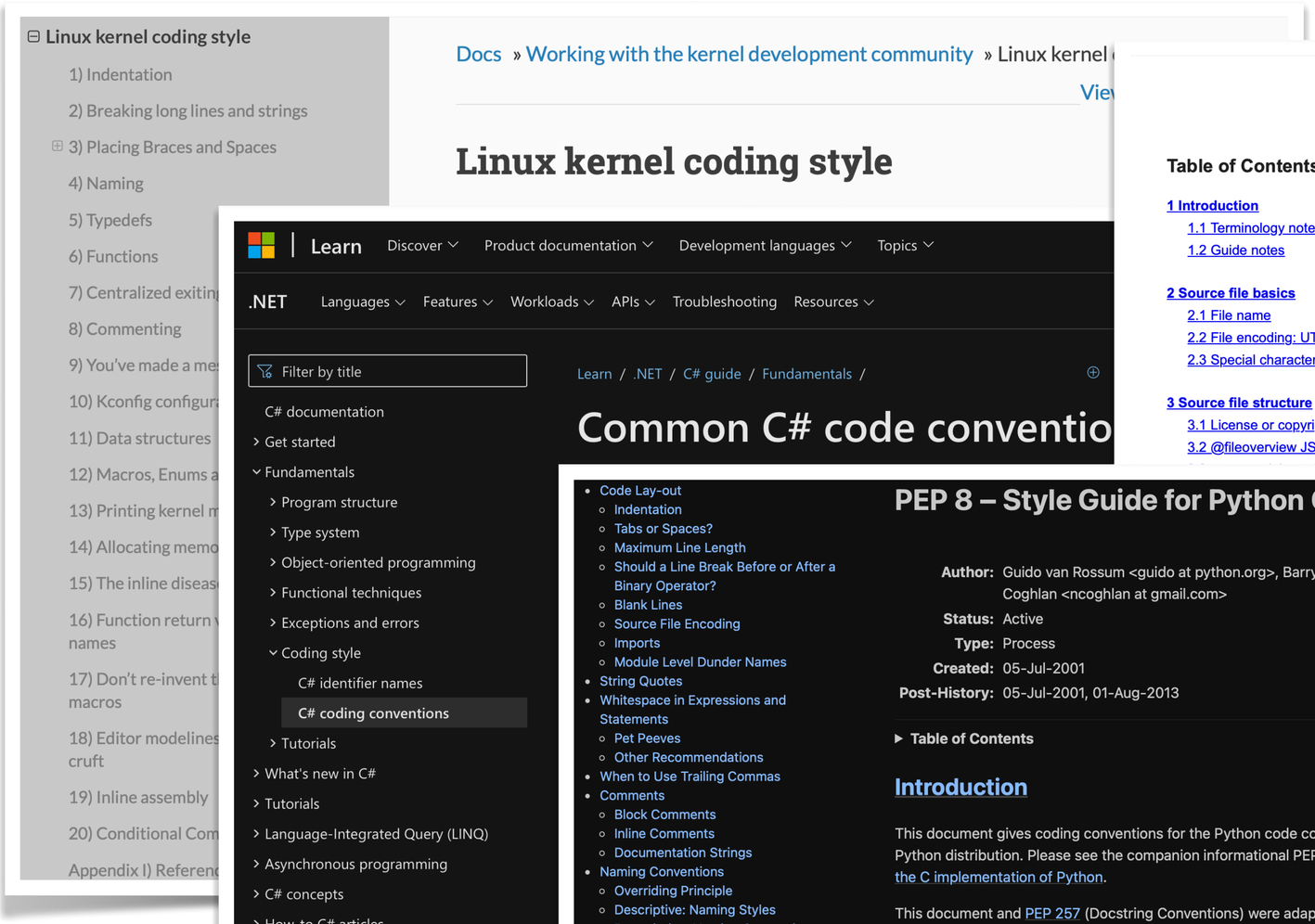
5 Language features
5.1 Local variable declarations
5.2 Array literals
5.3 Object literals
5.4 Classes
5.5 Functions
5.6 String literals
5.7 Number literals
5.8 Control structures
5.9 this
5.10 Equality Checks
5.11 Disallowed features

6 Naming
6.1 Rules common to all identifiers
6.2 Rules by identifier type
6.3 Camel case: defined

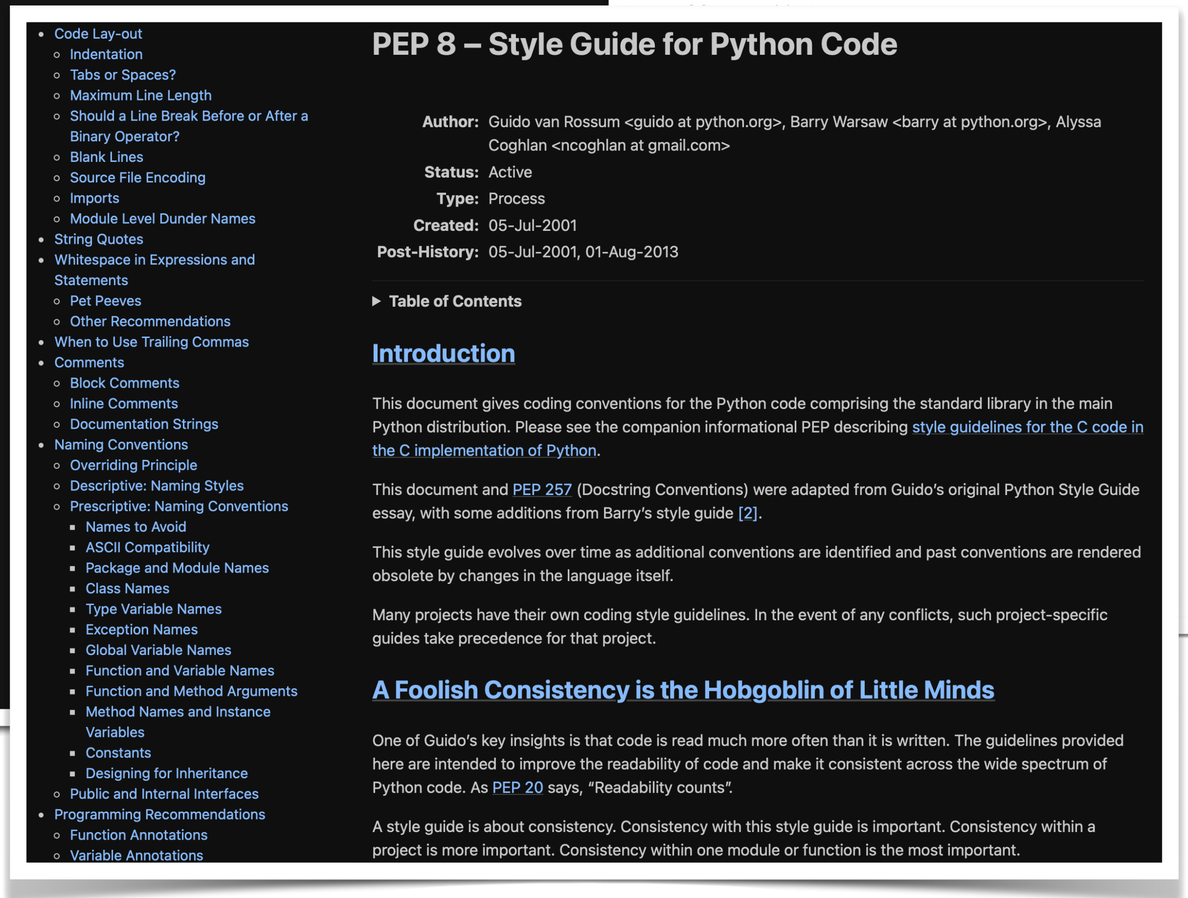
7 JSDoc
7.1 General form
7.2 Markdown
7.3 JSDoc tags
7.4 Line wrapping
7.5 Top/file-level comments
7.6 Class comments
7.7 Enum and typedef comments
7.8 Method and function comments
7.9 Property comments
7.10 Type annotations
7.11 Visibility annotations

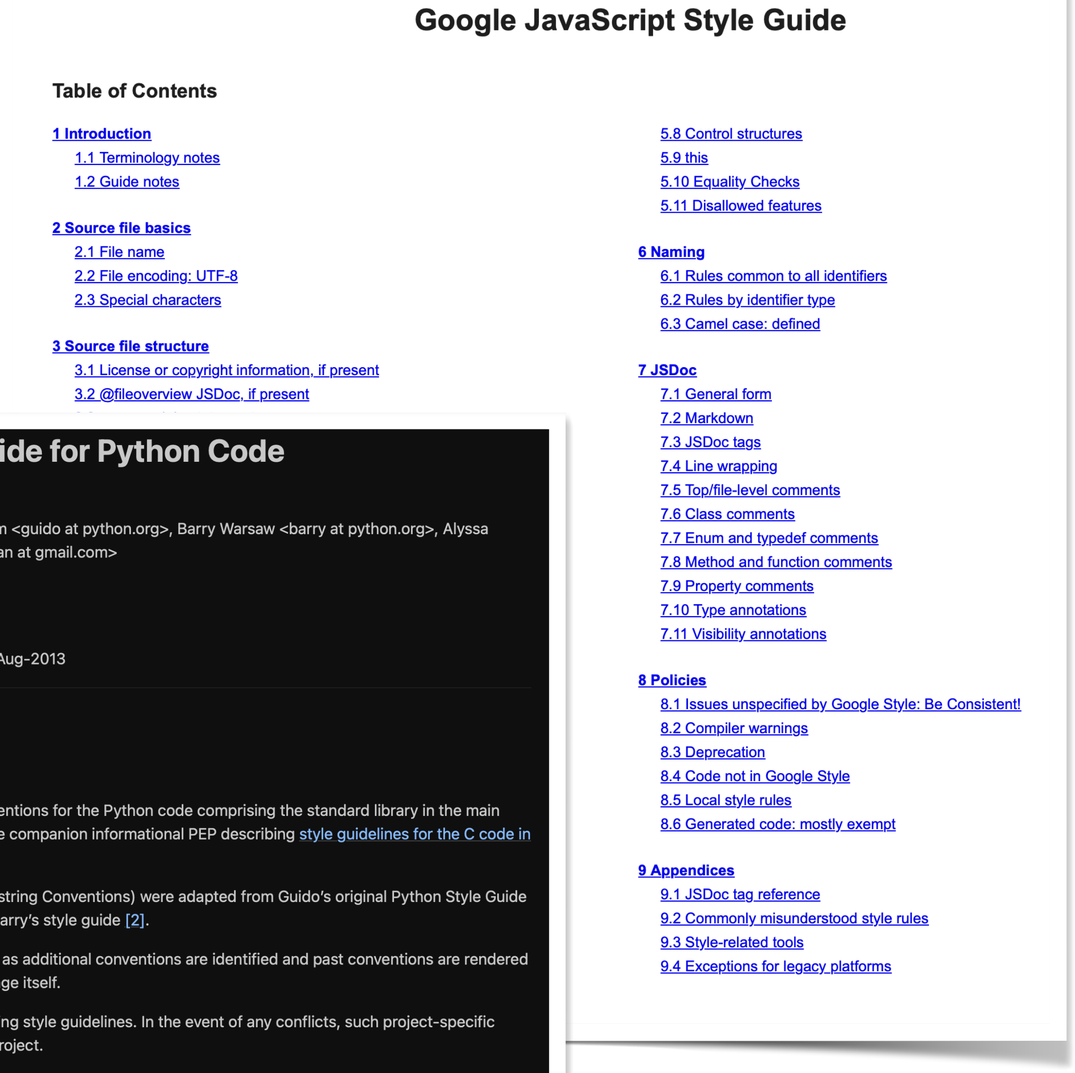
8 Policies
8.1 Issues unspecified by Google Style: Be Consistent!
8.2 Compiler warnings
8.3 Deprecation
8.4 Code not in Google Style
8.5 Local style rules
8.6 Generated code: mostly exempt

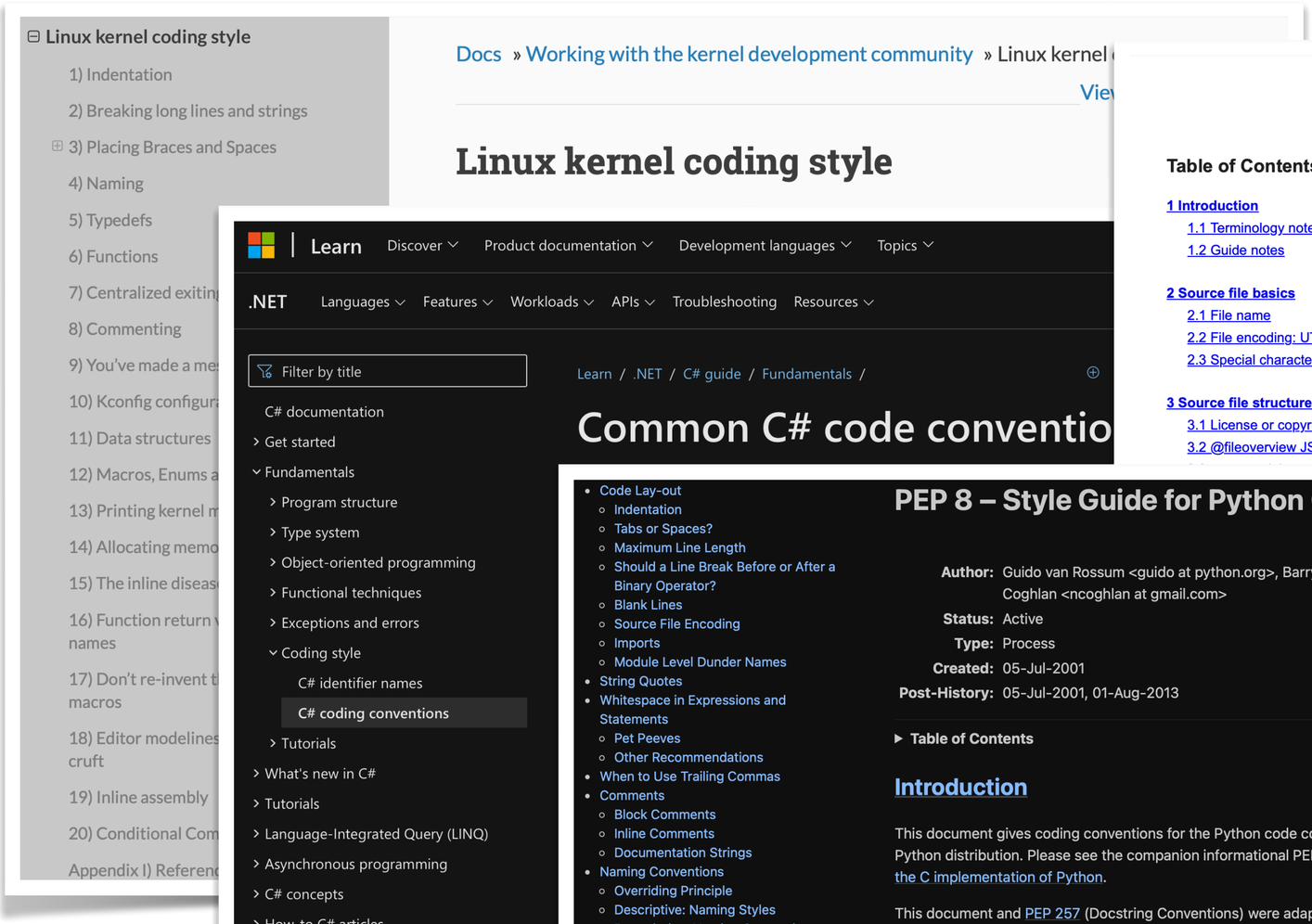
9 Appendices
9.1 JSDoc tag reference
9.2 Commonly misunderstood style rules
9.3 Style-related tools
9.4 Exceptions for legacy platforms


Linux kernel coding style
1) Indentation
2) Breaking long lines and strings
3) Placing Braces and Spaces
4) Naming
5) Typedefs
6) Functions
7) Centralized exiting
8) Commenting
9) You've made a mes
10) Kconfig configura
11) Data structures
12) Macros, Enums a
13) Printing kernel m
14) Allocating memo
15) The inline disease
16) Function return v
names
17) Don't re-invent t
macros
18) Editor modelines
cruft
19) Inline assembly
20) Conditional Com
Appendix I) Referenc

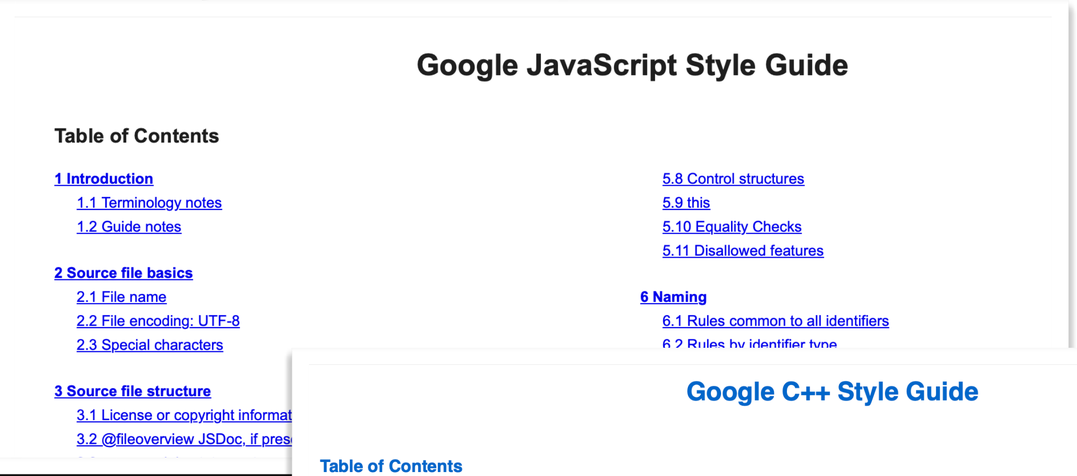
Docs » Working with the kernel development community » Linux kernel
Linux kernel coding style
Learn
Discover
Product documentation
Development languages
Topics
.NET
Languages
Features
Workloads
APIs
Troubleshooting
Resources
Filter by title
C# documentation
Get started
Fundamentals
Program structure
Type system
Object-oriented programming
Functional techniques
Exceptions and errors
Coding style
C# identifier names
C# coding conventions
Tutorials
What's new in C#
Tutorials
Language-Integrated Query (LINQ)
Asynchronous programming
C# concepts
How-to C# articles
Advanced topics
The .NET Compiler Platform SDK (Roslyn APIs)
C# programming guide
Language reference
Specifications
Learn / .NET / C# guide / Fundamentals /
Common C# code conventio


Code Lay-out
Indentation
Tabs or Spaces?
Maximum Line Length
Should a Line Break Before or After a Binary Operator?
Blank Lines
Source File Encoding
Imports
Module Level Dunder Names
String Quotes
Whitespace in Expressions and Statements
Pet Peeves
Other Recommendations
When to Use Trailing Commas
Comments
Block Comments
Inline Comments
Documentation Strings
Naming Conventions
Overriding Principle
Descriptive: Naming Styles
Prescriptive: Naming Conventions
Names to Avoid
ASCII Compatibility
Package and Module Names
Class Names
Type Variable Names
Exception Names
Global Variable Names
Function and Variable Names
Function and Method Arguments
Method Names and Instance Variables
Constants
Designing for Inheritance
Public and Internal Interfaces
Programming Recommendations
Function Annotations
Variable Annotations
PEP 8 – Style Guide for Python Code
Author: Guido van Rossum <guido at python.org>, Barry Warsaw <barry at python.org>, Alyssa Coghlan <ncoghlan at gmail.com>
Status: Active
Type: Process
Created: 05-Jul-2001
Post-History: 05-Jul-2001, 01-Aug-2013
Table of Contents
Introduction
This document gives coding conventions for the Python code comprising the standard library in the main Python distribution. Please see the companion informational PEP describing style guidelines for the C code in the C implementation of Python.
This document and PEP 257 (Docstring Conventions) were adapted from Guido's original Python Style Guide essay, with some additions from Barry's style guide [2].
This style guide evolves over time as additional conventions are identified and past conventions are rendered obsolete by changes in the language itself.
Many projects have their own coding style guidelines. In the event of any conflicts, such project-specific guides take precedence for that project.
A Foolish Consistency is the Hobgoblin of Little Minds
One of Guido's key insights is that code is read much more often than it is written. The guidelines provided here are intended to improve the readability of code and make it consistent across the wide spectrum of Python code. As PEP 20 says, "Readability counts".
A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is the most important.

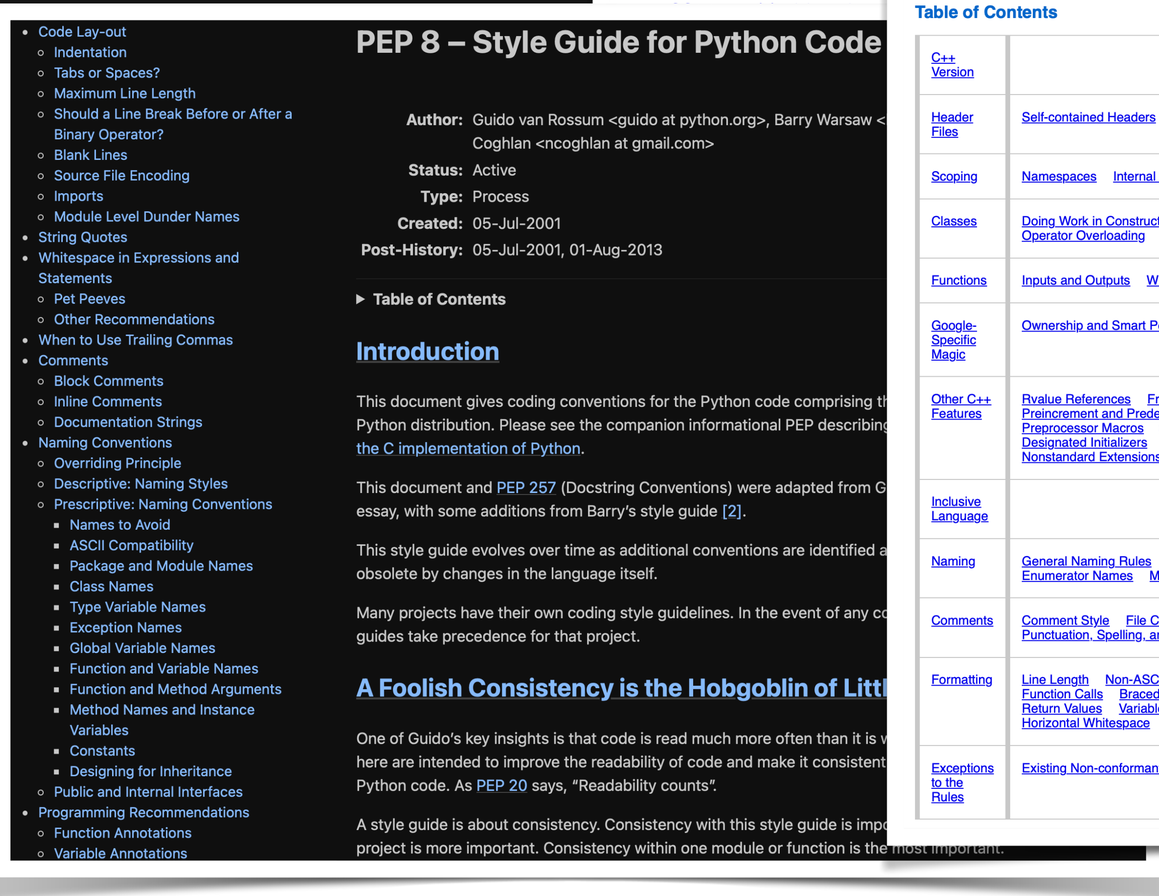

Google JavaScript Style Guide
Table of Contents
1 Introduction
1.1 Terminology notes
1.2 Guide notes
2 Source file basics
2.1 File name
2.2 File encoding: UTF-8
2.3 Special characters
3 Source file structure
3.1 License or copyright information, if present
3.2 @fileoverview JSDoc, if present
5.8 Control structures
5.9 this
5.10 Equality Checks
5.11 Disallowed features
6 Naming
6.1 Rules common to all identifiers
6.2 Rules by identifier type
6.3 Camel case: defined
7 JSDoc
7.1 General form
7.2 Markdown
7.3 JSDoc tags
7.4 Line wrapping
7.5 Top/file-level comments
7.6 Class comments
7.7 Enum and typedef comments
7.8 Method and function comments
7.9 Property comments
7.10 Type annotations
7.11 Visibility annotations
8 Policies
8.1 Issues unspecified by Google Style: Be Consistent!
8.2 Compiler warnings
8.3 Deprecation
8.4 Code not in Google Style
8.5 Local style rules
8.6 Generated code: mostly exempt
9 Appendices
9.1 JSDoc tag reference
9.2 Commonly misunderstood style rules
9.3 Style-related tools
9.4 Exceptions for legacy platforms

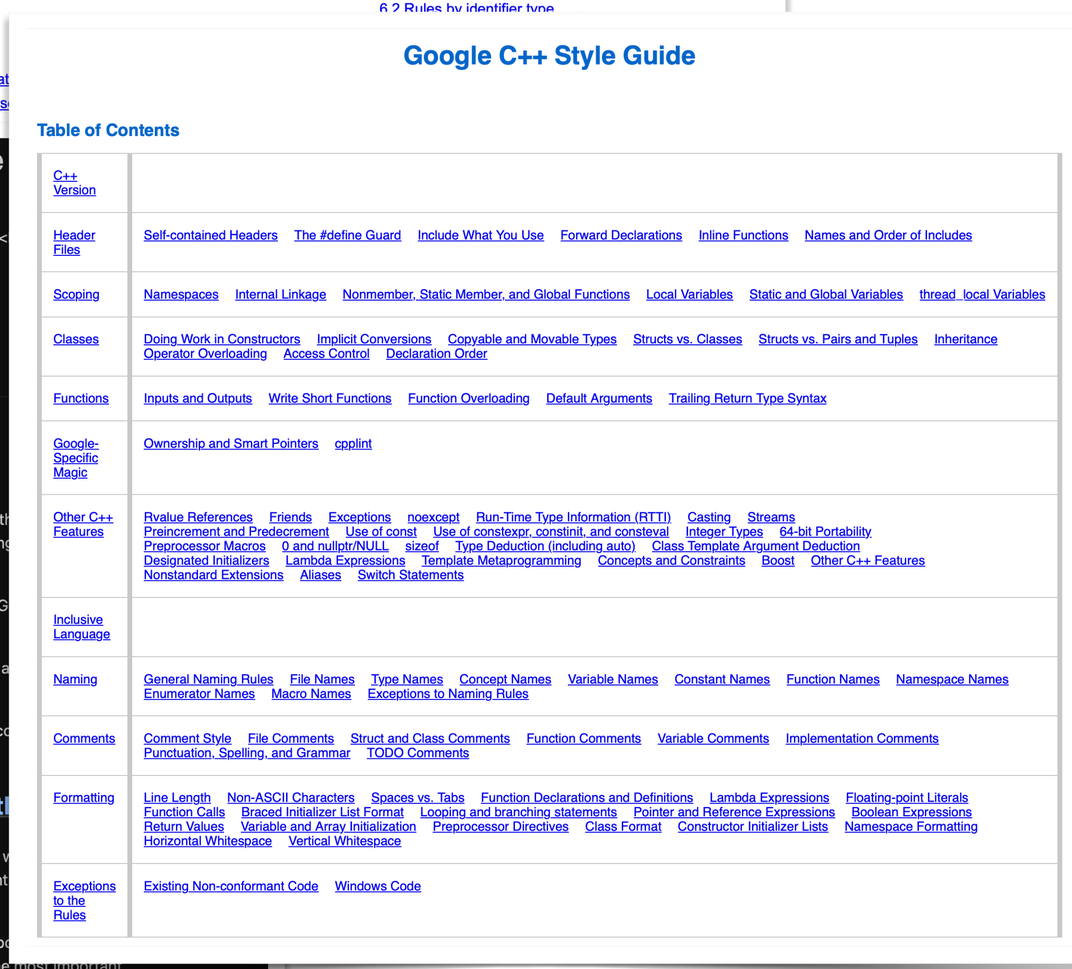

Linux kernel coding style
1) Indentation
2) Breaking long lines and strings
3) Placing Braces and Spaces
4) Naming
5) Typedefs
6) Functions
7) Centralized exiting
8) Commenting
9) You've made a me
10) Kconfig configura
11) Data structures
12) Macros, Enums a
13) Printing kernel m
14) Allocating memo
15) The inline disease
16) Function return v names
17) Don't re-invent t macros
18) Editor modelines cruft
19) Inline assembly
20) Conditional Com
Appendix I) Referenc

Docs » Working with the kernel development community » Linux kernel
Linux kernel coding style


Google JavaScript Style Guide
Table of Contents
1 Introduction
1.1 Terminology notes
1.2 Guide notes
2 Source file basics
2.1 File name
2.2 File encoding: UTF-8
2.3 Special characters
3 Source file structure
3.1 License or copyright informat
3.2 @fileoverview JSDoc, if pres
5.8 Control structures
5.9 this
5.10 Equality Checks
5.11 Disallowed features
6 Naming
6.1 Rules common to all identifiers
6.2 Rules by identifier type

Learn
Discover
Product documentation
Development languages
Topics
.NET
Languages
Features
Workloads
APIs
Troubleshooting
Resources
Filter by title
C# documentation
Get started
Fundamentals
Program structure
Type system
Object-oriented programming
Functional techniques
Exceptions and errors
Coding style
C# identifier names
C# coding conventions
Tutorials
What's new in C#
Tutorials
Language-Integrated Query (LINQ)
Asynchronous programming
C# concepts
How-to C# articles
Advanced topics
The .NET Compiler Platform SDK (Roslyn APIs)
C# programming guide
Language reference
Specifications
Learn / .NET / C# guide / Fundamentals /
Common C# code conventio


Code Lay-out
Indentation
Tabs or Spaces?
Maximum Line Length
Should a Line Break Before or After a Binary Operator?
Blank Lines
Source File Encoding
Imports
Module Level Dunder Names
String Quotes
Whitespace in Expressions and Statements
Pet Peeves
Other Recommendations
When to Use Trailing Commas
Comments
Block Comments
Inline Comments
Documentation Strings
Naming Conventions
Overriding Principle
Descriptive: Naming Styles
Prescriptive: Naming Conventions
Names to Avoid
ASCII Compatibility
Package and Module Names
Class Names
Type Variable Names
Exception Names
Global Variable Names
Function and Variable Names
Function and Method Arguments
Method Names and Instance Variables
Constants
Designing for Inheritance
Public and Internal Interfaces
Programming Recommendations
Function Annotations
Variable Annotations
PEP 8 – Style Guide for Python Code
Author: Guido van Rossum <guido at python.org>, Barry Warsaw <
Coghlan <ncoghlan at gmail.com>
Status: Active
Type: Process
Created: 05-Jul-2001
Post-History: 05-Jul-2001, 01-Aug-2013
Table of Contents
Introduction
This document gives coding conventions for the Python code comprising th
Python distribution. Please see the companion informational PEP describing
the C implementation of Python.
This document and PEP 257 (Docstring Conventions) were adapted from G
essay, with some additions from Barry's style guide [2].
This style guide evolves over time as additional conventions are identified a
obsolete by changes in the language itself.
Many projects have their own coding style guidelines. In the event of any c
guides take precedence for that project.
A Foolish Consistency is the Hobgoblin of Litt
One of Guido's key insights is that code is read much more often than it is v
here are intended to improve the readability of code and make it consistent
Python code. As PEP 20 says, "Readability counts".
A style guide is about consistency. Consistency with this style guide is impo
project is more important. Consistency within one module or function is the


Google C++ Style Guide
Table of Contents
C++ Version
Header Files
Self-contained Headers
The #define Guard
Include What You Use
Forward Declarations
Inline Functions
Names and Order of Includes
Scoping
Namespaces
Internal Linkage
Nonmember, Static Member, and Global Functions
Local Variables
Static and Global Variables
thread_local Variables
Classes
Doing Work in Constructors
Implicit Conversions
Copyable and Movable Types
Structs vs. Classes
Structs vs. Pairs and Tuples
Inheritance
Operator Overloading
Access Control
Declaration Order
Functions
Inputs and Outputs
Write Short Functions
Function Overloading
Default Arguments
Trailing Return Type Syntax
Google-Specific Magic
Ownership and Smart Pointers
cpplint
Other C++ Features
Rvalue References
Friends
Exceptions
noexcept
Run-Time Type Information (RTTI)
Casting
Streams
Preincrement and Predecrement
Use of const
Use of constexpr, constinit, and consteval
Integer Types
64-bit Portability
Preprocessor Macros
0 and nullptr/NULL
sizeof
Type Deduction (including auto)
Class Template Argument Deduction
Designated Initializers
Lambda Expressions
Template Metaprogramming
Concepts and Constraints
Boost
Other C++ Features
Nonstandard Extensions
Aliases
Switch Statements
Inclusive Language
Naming
General Naming Rules
File Names
Type Names
Concept Names
Variable Names
Constant Names
Function Names
Namespace Names
Enumerator Names
Macro Names
Exceptions to Naming Rules
Comments
Comment Style
File Comments
Struct and Class Comments
Function Comments
Variable Comments
Implementation Comments
Punctuation, Spelling, and Grammar
TODO Comments
Formatting
Line Length
Non-ASCII Characters
Spaces vs. Tabs
Function Declarations and Definitions
Lambda Expressions
Floating-point Literals
Function Calls
Braced Initializer List Format
Looping and branching statements
Pointer and Reference Expressions
Boolean Expressions
Return Values
Variable and Array Initialization
Preprocessor Directives
Class Format
Constructor Initializer Lists
Namespace Formatting
Horizontal Whitespace
Vertical Whitespace
Exceptions to the Rules
Existing Non-conformant Code
Windows Code