SUSE Best Practices

Development Tools

Introduction to the Python Library Pydantic

Pydantic
Python

Sushant Gaurav, Technical Writer (SUSE)

This document provides an introduction to Pydantic, a powerful Python library designed for data validation and settings management, and details its most important features.

Contents

1 Introduction

Data exchange is a fundamental part of modern applications, especially those that interact with APIs, databases, or external services. In Python (https://docs.python.org/3/)↗, ensuring data is structured, validated, and easily converted between formats is critical. This is where Pydantic comes into the picture.

Pydantic (https://docs.pydantic.dev/latest/)↗ is a powerful Python library designed for data validation and settings management. It uses Python type hints to define, parse, and enforce strict typing on data models (https://docs.pydantic.dev/latest/concepts/models/)↗ making it easier to work with structured and semi-structured data.

Whether you are handling user input, API responses, or configuration files, Pydantic ensures your application receives data in the correct format. If the data does not match the defined schema, Pydantic raises clear and informative validation errors. This helps reduce the debugging time and improves code reliability.

This document is intended to provide an initial introduction and overview of Pydantic. The following sections cover the most important functions and features of Pydantic. However, the document does not claim to be exhaustive.

1.1 Pydantic v1 vs. Pydantic v2: What changed?

Pydantic v2 represents a major upgrade over v1, offering improved performance, enhanced type safety, cleaner validation syntax, and better serialization tools. Although many v1 models will still work with v2, some migration steps (https://docs.pydantic.dev/latest/migration/)↗ are required due to changes in decorators and method names.

Key Differences at a Glance:

TABLE 1: COMPARISON: PYDANTIC V1 VS. PYDANTIC V2

Feature	Pydantic v1	Pydantic v2
Validation engine	Pure Python	Rust-based core (faster)
Field validators	`@validator`	`@field_validator`
Model validators	`@root_validator`	`@model_validator`
Computed fields	Via`@property`	Native via `@computed_field`

Feature	Pydantic v1	Pydantic v2
Type system	Standard type hints	Better support for advanced typing
Strict type handling	Limited	Enhanced (`StrictStr`, `StrictInt`, etc.)
Serialization	**`.dict()`**, **`.json()`**	**`_dump()`**, **`_dump_json()`**
Environment settings	`BaseSettings`	Improved support for config and env parsing
Error reporting	Simple	More structured and user-friendly

1.2 Major improvements in Pydantic v2

Improved Validation Flow

The introduction of `before` and `after` modes for both field and model validators gives developers precise control over the validation lifecycle. This enables early rejection of bad data or sophisticated cross-field logic as needed.

Enhanced Serialization

Pydantic v2's **`.model_dump()`** and **`.model_dump_json()`** provide a more flexible and consistent serialization API, especially when dealing with computed fields or nested models.

2 The role of `BaseModel`

The core of Pydantic is the `BaseModel` (https://docs.pydantic.dev/latest/api/base_model/) class. Every custom model in Pydantic is built by inheriting this base class. It provides essential functionality, including:

- Field definition and type enforcement
- Built-in validation and error messaging

- Data serialization and deserialization
- Automatic type conversion when possible

By extending `BaseModel`, you gain access to a robust set of tools that ensure your data conforms to the expected types and structures.

Example (Product class inherits `BaseModel` class):

```
from pydantic import BaseModel

class Product(BaseModel):
price: int
```

2.1 Automatic type conversion

One of the most useful features of Pydantic is its ability to perform type coercion automatically. When *possible*, Pydantic converts input data to match the declared type.

Example:

```
python
from pydantic import BaseModel

class Product(BaseModel):
   price: int

Product(price='99')       # Allowed: '99' is converted to 99
Product(price='99a')      # Raises ValidationError: cannot convert to int
```

Error:

```
Product(price='99a')      # Raises ValidationError: cannot convert to int
 File "/Users/imsushant/Library/Python/3.9/lib/python/site-packages/pydantic/main.py",
 line 253, in __init__
   validated_self = self.__pydantic_validator__.validate_python(data, self_instance=self)
pydantic_core._pydantic_core.ValidationError: 1 validation error for Product
price
 Input should be a valid integer, unable to parse string as an integer [type=int_parsing,
 input_value='99a', input_type=str]
   For further information visit https://errors.pydantic.dev/2.11/v/int_parsing
```

In this example, a string containing numeric characters (`'99'`) is accepted and converted to an integer, while an invalid string (`'99a'`) results in a validation error.

2.2 Real-world example (Parsing JSON)

Consider a scenario where your application receives JSON data from an external API:

```
{
 "id": "101",
 "name": "Sam",
 "salary": "12000"
}
```

You can use Pydantic to model and validate this data:

```
from pydantic import BaseModel

class Employee(BaseModel):
   id: int
   name: str
   salary: float

emp = Employee(**json_data)
print(emp)
# Employee(id=101, name='Sam', salary=12000.0)
```

Note: Automatic conversion

Even though the `id` and `salary` fields arrive as strings, Pydantic automatically converts them to the appropriate numeric types.

If you do not want or like the automatic conversion, Pydantic allows to enable the strict feature. Read more in *Section 3.1, "Enforcing strict types with `Field(strict=True)`"*.

In the above example, the `**` operator is unpacking the `json_data` dictionary into keyword arguments.

3 Understanding `Field` function

In Pydantic, `Field` (https://docs.pydantic.dev/latest/api/fields/) is used to configure individual model attributes. It allows you to define default values, add validation constraints, and provide metadata such as titles, descriptions, or example values.

It is not mandatory but using `Field` helps improve clarity, maintainability, and control over model behavior.

The key use cases for `Field` include:

- Defining default values
- Setting validation constraints (such as minimum or maximum length)
- Adding metadata (like descriptions or usage examples)
- Customizing serialization or documentation behavior

The common parameters used with `Field` include:

- `default`, `title`, `description`, `examples`
- `min_length`, `max_length`, `regex`
- `ge` (greater than or equal to), `le` (less than or equal to)
- `strict`, `frozen`, and others

Example:

```
from pydantic import BaseModel, Field, PositiveInt

class User(BaseModel):
   id: int
   name: str = Field(min_length=3)
   age: PositiveInt = Field(default=18)
```

Here, `name` must be at least three characters long, and `age` must be non-negative, with a default value of 18.

Note: Keep in mind:

- To mark a field as optional, wrap the type with `Optional[...]`.
- Use `...` (Ellipsis) to mark a field as required when no default is provided.
- `PositiveInt` is a Pydantic type that automatically enforces the constraint that the integer must be positive.

3.1 Enforcing strict types with `Field(strict=True)`

By default, Pydantic tries to coerce values into the expected type. This behavior is helpful in many situations. But, in some scenarios, you may need stricter validation for sensitive or critical fields. For such scenarios, use `strict=True` to enforce exact type matching:

Example:

```
from pydantic import BaseModel, Field

class User(BaseModel):
   age: int = Field(strict=True)

User(age="21")  # Raises validation error: str is not int
```

4 Using `Annotated` for cleaner type definitions

The `Annotated` type was introduced in the `typing` module to allow additional metadata to be attached to a type hint. Pydantic v2 adopts `Annotated` to define constraints, descriptions, and field-level metadata in a more structured and expressive way.

Learn more about the `typing` module at https://docs.python.org/3/library/typing.html ↗.

Reasons to use `Annotated`:

- Avoids mixing logic between type declaration and field definition
- Keeps the syntax cleaner and easier to read
- Enhances compatibility with tools like https://fastapi.tiangolo.com/ ↗ `FastAPI`that generate documentation from model metadata

Example: Classic `Field` usage without `Annotated`

```
from pydantic import BaseModel, Field

class Patient(BaseModel):
   name: str = Field(
       title="Patient Name",
       description="It contains the name of the patient",
       examples=["Aman", "Suman"]
   )
```

Example: Modern usage with `Annotated`

```python
from typing import Annotated
from pydantic import BaseModel, Field

class Patient(BaseModel):
   name: Annotated[
       str,
       Field(
           title="Patient Name",
           description="It contains the name of the patient",
           examples=["Aman", "Suman"]
       )
   ]
```

Example: Combining constraints and metadata

This approach ensures clean definitions, enforces constraints, and provides rich metadata for tools and documentation.

```python
from typing import Annotated
from pydantic import BaseModel, Field

class Patient(BaseModel):
    name: Annotated[
       str,
       Field(
           title="Patient Name",
           description="It contains the name of the patient",
           min_length=2,
           examples=["Aman", "Suman"]
       )
   ]
   age: Annotated[
       int,
       Field(
           ge=0,
           description="Patient age (non-negative)"
       )
   ]
```

5 Using validators and `ValidationError` in Pydantic

Pydantic supports custom data validation through **validators** (https://docs.pydantic.dev/latest/concepts/validators/)↗. Validators provide fine-grained control over how fields or models are validated, making them useful when the built-in validation logic is insufficient.

Pydantic includes two primary types of validators:

- field validators (https://docs.pydantic.dev/latest/concepts/validators/#field-validators) and
- model validators (https://docs.pydantic.dev/latest/concepts/validators/#model-validators)

5.1 Field validators

Field-level validators allow you to define custom logic for individual fields. These validators are declared using the `@field_validator` decorator. Introduced in Pydantic v2, this decorator must be used in combination with Python's standard `@classmethod` decorator.

Example:

```
from pydantic import BaseModel, field_validator

class User(BaseModel):
   username: str

   @field_validator("username")
   @classmethod
   def validate_username(cls, value):
       if " " in value:
          raise ValueError("Username must not contain spaces")
      return value
```

In the above example:

- `@classmethod` is applied first.
- `field_validator` decorator contains the field name to validate.
- `cls` refers to the model class (`User`).
- `value` is the input provided for the `username` field.

Note: Correct order of decorators

The correct order of decorators is crucial. The `@classmethod` decorator must be applied before the `@field_validator` decorator, as Pydantic v2's `@field_validator` expects a class method (not a static method or instance method).

You can even apply a single validator to multiple fields.

Example:

```
from pydantic import BaseModel, field_validator

class User(BaseModel):
   username: str
   email: str

   @field_validator("username", "email")
   @classmethod
   def no_spaces_allowed(cls, value, info):
       if " " in value:
           raise ValueError(f"{info.field_name} must not contain spaces")
       return value
```

In the above example, the `no_spaces_allowed()` function is executed once per field, and the `info` parameter provides metadata such as the current field name.

Note: Field validators

Field validators execute before type coercion by default, which makes them ideal for validating raw input values.

5.2 Model validators

Model-level validators are methods used to validate the entire data model at once. They are useful for performing cross-field validation, where the validation of one field depends on the value of another, or for complex checks that involve multiple fields. In Pydantic v2, you use the `@model_validator` decorator for this purpose. This decorator replaces the earlier `@root_validator` from Pydantic v1.

```
from pydantic import BaseModel, model_validator

class User(BaseModel):
   password: str
   confirm_password: str

   @model_validator(mode="after")
   def passwords_match(self):
       if self.password != self.confirm_password:
           raise ValueError("Passwords do not match")
       return self
```

In the above example, `self` resembles the instance of the model.

5.3 Validator modes (`before` and `after`)

You can control when a validator runs using the `mode` argument:

- `mode="before"` executes before standard Pydantic validation.
- `mode="after"` executes after Pydantic validation and type coercion.

TABLE 2: VALIDATOR TYPES

Validator Type	`before` **Mode**	`after` **Mode (Default)**
Field Validator	Receives raw input	Receives parsed and type-coerced value
Model Validator	Receives raw input dictionary	Receives fully validated model instance

In short, use `"before"` when you need to clean or reject invalid data early. Use `"after"` for final checks, consistency validation, or business logic after all fields have been validated.

5.4 Key differences between field and model validator

Although `@field_validator("username", "email")`` accepts multiple fields, it still processes each field **independently**. This means that the validator runs separately for each field, even if the logic is identical. This approach can lead to redundant processing and is not ideal for validations that depend on multiple fields. In such cases, using a `@model_validator` is more efficient and appropriate, as it processes the entire model at once and allows for cross-field validation in a single pass.

TABLE 3: DIFFERENCES BETWEEN VALIDATOR TYPES

Aspect	**Field Validator**	**Model Validator**
Execution	Per field	Once per model instance
Input	Single field value (+ metadata)	Entire model (either raw or parsed)

Aspect	Field Validator	Model Validator
Use Case	Field-level validation and transformation	Cross-field validation, business logic

5.5 Handling validation errors

Whenever validation fails, Pydantic raises a `ValidationError` (https://docs.pydantic.dev/latest/concepts/validators/#raising-validation-errors). This exception provides a detailed breakdown of the issue, including the field name, the error message, and the error type.

Example:

```
from pydantic import BaseModel, ValidationError

class Product(BaseModel):
   price: float

try:
   Product(price="free")
except ValidationError as e:
   print(e)

# Output:
# 1 validation error for Product
# price
#   Input should be a valid number (type=type_error.float)
```

To programmatically inspect errors, you can use `e.errors()` which returns a list of structured error dictionaries.

6 Dumping model data in Pydantic

Pydantic models provide two helpful methods to extract data:

- **`.model_dump()`** (https://docs.pydantic.dev/latest/concepts/serialization/#modelmodel_dump) returns the model as a standard Python `dict`.
- **`.model_dump_json()`** (https://docs.pydantic.dev/latest/concepts/serialization/#modelmodel_dump_json) returns the model data as a JSON string.

These methods are commonly used when serializing models for storage, logging, or API responses.

Example:

```
from pydantic import BaseModel

class User(BaseModel):
   name: str
   age: int

user = User(name="Sam", age=25)

# model_dump() returns a Python dictionary
dumped_user = user.model_dump()

print(dumped_user)
# Output: {'name': 'Sam', 'age': 25}

print(type(dumped_user))
# Output: <class 'dict'>

# model_dump_json() returns a JSON-formatted string
json_user = user.model_dump_json()

print(json_user)
# Output: {"name":"Sam","age":25}

print(type(json_user))
# Output: <class 'str'>
```

To learn about these methods in detail, check *Section 9, "Understanding serialization in Pydantic"*.

7 Understanding computed fields in Pydantic

In many applications, certain values are not provided directly by the user but are derived from other fields. These values are known as **computed fields** (https://docs.pydantic.dev/2.0/usage/computed_fields/).

Pydantic v2 provides support for computed fields through the `@computed_field` decorator. This eliminates the need for workarounds, such as using `@property`, and provides better integration with Pydantics Serialization (https://docs.pydantic.dev/latest/concepts/serialization/) system.

Example:

```
from pydantic import BaseModel, computed_field

class Rectangle(BaseModel):
     width: float
     height: float

    @computed_field
    def area(self) -> float:
          return self.width * self.height
```

In the above example:

- area is computed dynamically based on width and height.
- the `@computed_field` decorator registers the method as a virtual field.

Note: Not included by default

Computed fields are not included in serialized outputs by default, preserving a clear boundary between user-provided input and derived values.

7.1 Including Computed Fields in Output

By default, computed fields are excluded from methods such as **`.model_dump()`** and **`.model_dump_json()`**. To include them, use the `include_computed=True` argument:

```
rect = Rectangle(width=10, height=5)
      print(rect.model_dump())
      # Output: {'width': 10, 'height': 5}

      print(rect.model_dump(include_computed=True))
      # Output: {'width': 10, 'height': 5, 'area': 50}
```

This behavior is intentional as it ensures that:

- computed values are not accidentally persisted or sent over APIs.
- only explicitly requested values are included.
- expensive calculations are avoided unless needed.

8 Working with nested models in Pydantic

Pydantic supports nested models, enabling developers to represent structured, hierarchical data in a clean and intuitive way. This is particularly useful in applications that deal with complex schemas such as user profiles, blog posts, or nested comment threads (where one model depends on or contains another).

Nested models in Pydantic allow you to:

- Represent relationships like `UserProfile -> Address -> Country`
- Encapsulate multi-level structured data (for example, `Blog -> Comment -> Author`)
- Validate recursive structures (for example, trees, chat threads)

Pydantic offers three main types of nested model patterns, which are detailed in the next sections.

8.1 Standard nesting (referencing other models)

The most common way to create nested data structures in Pydantic is through **composition**. This involves referencing one Pydantic model inside another using type annotations. This approach allows you to build layered schemas, where a complex model is composed of simpler, reusable components, and it ensures each part is validated correctly.

```
class Lesson(BaseModel):
   title: str
   content: str

class Module(BaseModel):
   name: str

   lessons: List[Lesson]  # Nested model: referencing Lesson inside Module
```

This setup allows each `Module` to encapsulate a list of validated `Lesson` instances, promoting reusability and data consistency.

8.2 Self-referencing models (recursive structures)

Pydantic also supports models that reference themselves. These are especially useful for representing recursive structures such as tree hierarchies or nested folders.

```
class Node(BaseModel):
   name: str
   children: List['Node']  # Forward reference using a string

Node.model_rebuild()
```

Because the class refers to itself and has not yet been fully constructed at the time of annotation, Pydantic requires **`model_rebuild()`** to resolve the forward reference. This ensures the `children` field is properly typed for validation and schema generation.

8.3 Forward referencing between multiple models

For mutually dependent models where, for example, an `Employee` references a `Manager`, and the `Manager` holds a list of `Employee` instances, Pydantic supports forward references using string annotations.

```
class Employee(BaseModel):
   name: str
   manager: 'Manager' = None  # String-based forward reference

class Manager(BaseModel):
   name: str
   team: List[Employee] = []

# Resolve circular references
Employee.model_rebuild()
Manager.model_rebuild()
```

Without **`model_rebuild()`**, these forward references would remain unresolved and lead to validation errors or incorrect schema generation.

8.4 Model inheritance

Pydantic models can also be extended using class inheritance, following standard Python principles. This pattern is useful for creating specialized models that share common fields and behavior with a parent model.

Example:

```
from pydantic import BaseModel

class Person(BaseModel):
   name: str
```

```
    age: int

class Worker(Person):
    company: str
    team: str

# Creating an instance of the derived model
worker = Worker(name="Alice", age=30, company="TechCorp", team="Engineering")

print(worker.model_dump())
# Output: {'name': 'Alice', 'age': 30, 'company': 'TechCorp', 'team': 'Engineering'}
```

In this example, the `Worker` model inherits the `name` and `age` fields from the `Person` model and adds its own unique fields, `company` and `team`. This approach promotes code reuse and helps maintain consistency across related models.

While inheritance is a powerful feature, Pydantic generally recommends using **composition** (as seen in standard nesting) over inheritance in most cases. Composition typically leads to more flexible and loosely coupled code, which can be easier to maintain and extend in the long run.

8.4.1 Importance of `model_rebuild()`

When forward references are used (either to the same model or another model that has not been defined yet), **`model_rebuild()`** is necessary to:

- re-evaluate type hints that were expressed as strings.
- replace string references with actual class objects.
- finalize model field definitions for accurate parsing and validation.

This post-definition step allows Pydantic to maintain correctness in complex model structures.

8.5 Example: Nested Comment model

The following is an example of a nested `Comment` model as commonly found in blog platforms. It incorporates all three nested model concepts:

- Referencing an `Author` model
- Self-referencing through nested replies
- Forward referencing using strings

```
from pydantic import BaseModel
from typing import List, Optional

class Author(BaseModel):
   user_id: int
   username: str

class Comment(BaseModel):
   comment_id: int
   content: str
   author: Author  # Standard nesting
   replies: Optional[List['Comment']] = None  # Self-referencing

Comment.model_rebuild()

author1 = Author(user_id=1, username="Sam")

reply1 = Comment(
   comment_id=2,
   content="Replying to your comment!",
   author=author1
)

main_comment = Comment(
   comment_id=1,
   content="This is the main comment",
   author=author1,
   replies=[reply1]
)

print(main_comment.model_dump(indent=2))
```

Output:

```
{
 "comment_id": 1,
 "content": "This is the main comment",
 "author": {
   "user_id": 1,
   "username": "Sam"
 },
 "replies": [
   {
     "comment_id": 2,
     "content": "Replying to your comment!",
     "author": {
       "user_id": 1,
```

```
      "username": "Sam"
    },
    "replies": null
  }
 ]
}
```

9 Understanding serialization in Pydantic

Serialization (https://docs.pydantic.dev/latest/concepts/serialization/) in Pydantic refers to the process of converting a model into a format suitable for storage or transmission.

This typically means converting a model to a dictionary (`dict`dict) or a JSON string (`str`).

Pydantic also handles deserialization, which converts raw input data into structured model instances.

Pydantic supports two key directions in serialization, which are detailed below.

9.1 Up derialization (deserialization)

Up serialization, also known as deserialization, is the process of converting raw data such as a dictionary or JSON object into a Pydantic model. This enables structured handling of unstructured input data.

Example:

```
data = {"name": "Sam", "joined": "2023-07-24T10:00:00"}
user = User(**data)  # Deserializes raw dict into a User model
```

9.2 Down serialization

Down serialization refers to converting a Pydantic model into a format like a Python dictionary or a JSON string. This is typically used when the model data needs to be sent over a network, saved to a file, or logged.

9.3 Methods for serialization

9.3.1 `model_dump()`

The **`model_dump()`** method converts a Pydantic model into a standard Python dictionary It excludes computed fields by default unless `include_computed=True` is specified.It is best suited for internal Python logic, debugging, or when the data remains within the application.

9.3.2 `model_dump_json()`

The **`model_dump_json()`** method returns a JSON-formatted string. It automatically converts non-JSON-native types (such as `datetime` or `Decimal`) into JSON-compatible representations, such as ISO 8601 strings. It is ideal for API communication, file storage, or external logging.

Example with `datetime`:E

```
from pydantic import BaseModel
from datetime import datetime

class User(BaseModel):
   name: str
   joined: datetime

# Up serialization: Create model from raw data
user = User(name="Sam", joined="2023-07-24T10:00:00")

# Down serialization: Convert to dictionary
print(user.model_dump())
# Output: {'name': 'Sam', 'joined': datetime.datetime(2023, 7, 24, 10, 0)}

# Down serialization: Convert to JSON string
print(user.model_dump_json())
# Output: {"name": "Sam", "joined": "2023-07-24T10:00:00"}
```

10 FAQs

10.1 How does field aliasing affect serialization in Pydantic?

Pydantic supports aliasing field names using the `alias` parameter in the `Field()` function. This is particularly useful when you need to conform to naming conventions, such as camelCase in API responses.

```
from pydantic import BaseModel, Field

class User(BaseModel):
   full_name: str = Field(..., alias="fullName")

user = User(fullName="Sam")
print(user.model_dump(by_alias=True))  # {'fullName': 'Sam'}
```

To ensure aliases are included in the serialized output, set `by_alias=True` when calling **`model_dump()`** or **`model_dump_json()`**.

10.2 Are `model_dump()` and `model_dump_json()` available in Pydantic v1?

No. These methods are part of **Pydantic v2**. In Pydantic v1, serialization was handled using the `dict()` and `json()` methods.

If you're upgrading from v1 to v2, note the following changes:

- **`model.dict()`** **`model.model_dump_json()`**

10.3 What happens if a field contains a non-serializable object during `model_dump_json()`?

If a field contains a non-JSON-serializable object, such as a custom class or complex data type, **`model_dump_json()`** will raise a `TypeError`. To handle this, you can either preprocess the data or define custom `json_encoders` in your model's configuration.

```
class Config:
   json_encoders = {
```

```
        CustomType: lambda v: str(v)
    }
```

10.4 Can I exclude certain fields when using **model_dump()** or **model_dump_json()**?

Yes. Both methods support parameters like `exclude`, `include`, and `exclude_unset`.

```
model.model_dump(exclude={"password"})
```

These options help you control exactly what gets serialized. It is useful when returning partial responses or removing sensitive data like passwords or tokens.

10.5 Is it possible to serialize nested models using **model_dump()** or **model_dump_json()**?

Yes. Nested models are fully supported. When serialized, they are recursively converted to dictionaries or JSON strings.

```
class Address(BaseModel):
    city: str

class User(BaseModel):
   name: str
   address: Address

user = User(name="Sushant", address=Address(city="Delhi"))
print(user_dump())
# {'name': 'Sushant', 'address': {'city': 'Delhi'}}
```

10.6 How to print compact or pretty-printed JSON output using **model_dump_json()**?

By default, **`model_dump_json()`** produces compact JSON. If you want pretty-printed (indented) output, you can pass formatting arguments using the `indent` parameter.

```
user_dump_json(indent=2)
```

11 Legal notice

GNU Free Documentation License

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/ ↗.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

```
Copyright (c) YEAR YOUR NAME.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is included in the section entitled "GNU
 Free Documentation License".
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts". line with this:

```
with the Invariant Sections being LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.