Designing Clean Interfaces

SECT.06.2026

- What constitutes good software *interface design*?
- What are some common interface patterns that can improve our code?

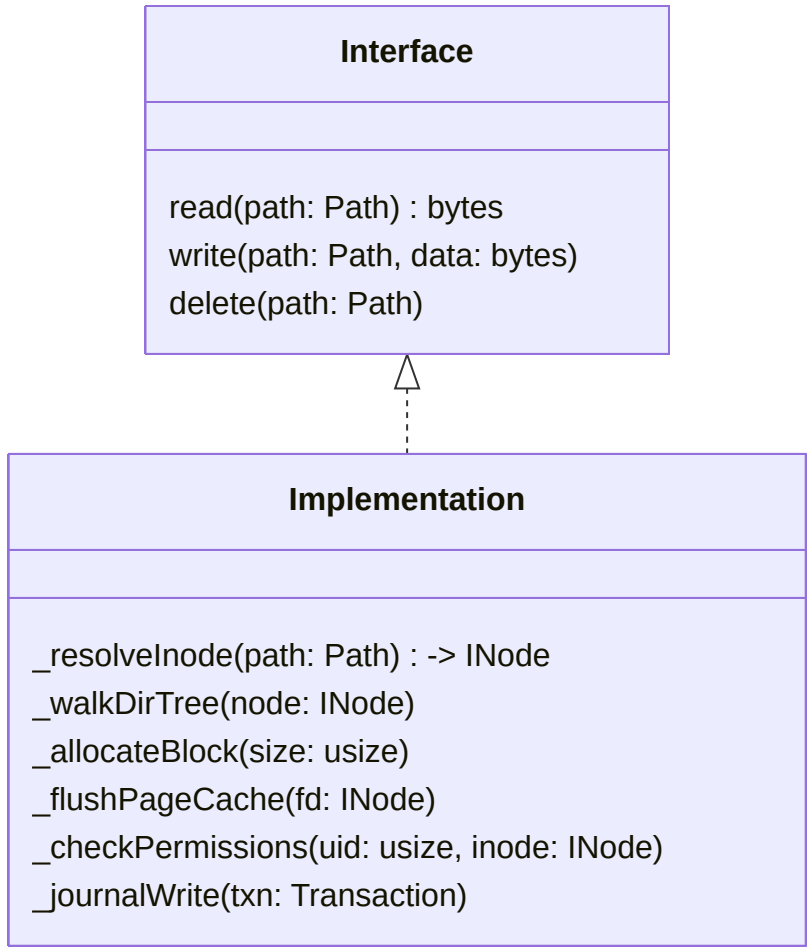
Modules/Packages

A **package** is made up of **functions** and **types: behavior** and **data**.

In Python 'module' formally refers to a file with many modules forming a 'package'.

In the broader sense we think of modules as groupings of behaviors.

The **interface** is the portion of the module intended to be used by code *outside the module*.



How to Divide?

Previously, perhaps *who is working on it?*

Ideally, *what does it do?*

In a well-designed project, if I am looking for a piece of functionality you have a good idea where to look even if I've never been in that part of the code before.

Beware complexity via over-modularization: `N` modules means `N^2` possible connections.

Layers

It is helpful to think in *layers*: UI, business logic, auth, data storage, etc.

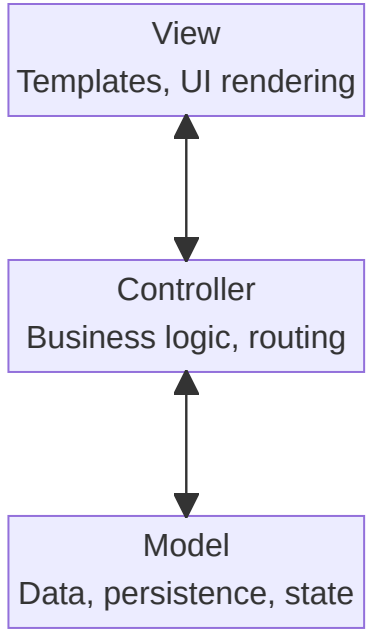
Each layer can only access layers "beneath" it (ideally just the layer(s) immediately beneath)

Before the layer is written, there are tons of possibilities.

How will this application store user data?

The data storage layer contains these choices, ideally all code related to our database (or S3, etc.) is contained in this layer. *Why?*

The delegated choices are exposed the **interface**.



Interfaces Provide...

Encapsulation

Limit complexity to a single portion of the code.

An HTTP module should not need to know if the underlying network is wired or wireless.

Hedge for multiple implementations

`check_login(username, password)` - Can continue to work when we switch from sqlite to MongoDB to Postgres.

Testing/Mocking

Allows *comprehensible* units that a single person can review & evaluate for correctness.

Space to think about design needs

Interface design encourages us to think about what the rest of the program might need from us.

Considered future needs can influence implementation. (but balance w/ YAGNI)

Interfaces in Python

- `__dunder__` methods: provide common interfaces for mapping Python syntax onto our classes
- ABCs: Iterable, Callable, Sequence, Mapping
- `load/dump`

Collection

```
class MyCollection:
  def __contains__(self, index): ... # item in myseq
  def __iter__(self):  ... # for item in myseq
  def __len__(self):  ... # len(myseq)
```

Sequence

```
class MySequence:
  def __contains__(self, index): ...
  def __iter__(self):  ...
  def __len__(self):  ...
  def __getitem__(self, index): ... # myseq[index]
```

Mapping

Adds `keys`, `values`, `items`, `get`, etc.

load/dump

```
# (approx.)
dump(data: list | dict, file: SupportsRead) → None
load(fp: SupportsWrite) → list | dict
dumps(data: list | dict) → str
loads(s: str | bytes) → list | dict
```

implementations

```
import json
json.dump(data, file)
data = json.load(file)

import yaml
yaml.dump(data, file)
data = yaml.load(file)

import pickle
pickle.dump(data, file)
data = pickle.load(file)
```

interchangeable parts

```
import json as filefmt

filefmt.dump(data, file)
```

```
# compressed.py
def dump(data, file):
   ...
def load(file) -> data:
   ...
```

```
import compressed as filefmt

filefmt.dump(data, file)
```

Principles of Interface Design

Single Responsibility Principle

Encapsulation vs. Leaky Abstractions

Simple Interface > Simple Implementation

Consistency

Set of Primitives

Single Responsibility Principle

"A module should be responsible to one, and only one, actor"

- Hard to name? Split the module
- Similar/identical signatures? *Consider* sharing module.
- Avoid (large) `utils` modules: the "junk drawer" of modules.

```
def load_csv(filename):
   ...

def load_json(filename):
   ...

def count_followers(user):
   ...

def handle_ui_click( ... ):
   ...

def run_db_report(db):
   ...
```

Encapsulation

Depend on interface, not on implementation.

Leaky abstractions make it tempting, or even necessary to depend on implementation.

Leaky Function

```
def parse_markdown(
    file: pathlib.Path,
        bold_marker: str,
        em_marker: str,
        code_block_marker: str,
    )

def _marker_to_regex(str) -> regex:
    return re.compile("\b{str}(.+?){str}\b")

parse_markdown( ... , "**", "*", "`")
```

What happens if function changes to using a non-regex based parser?

Leaky Class

Typically reveal attributes:

```
tbl = Table( ... )

for header in tbl.headers():
        ...

tbl.replace_column("col_name", [1,2,3])

# could also use __setitem__
tbl["col_name"] = [1, 2, 3]
```

What changes break this?

Other problems with exposing this data?

Simple Interface > Simple Implementation

It is often worth adding *some* complexity to your implementation in the name of a better interface.

Example

```
qs = Model.objects.filter( ... ) # QuerySet

# almost all QuerySet methods return a new QuerySet
qs = qs.filter( ... ).exclude( ... ).values( ... )

# no DB query done so far in this code

# only when a QuerySet is converted to another type
# (e.g. to strings for template) would the query be performed
list(qs)
```

Complex implementation worth it because it enables much simpler code higher up in the stack.

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Consistency

Care should be taken to make the API internally consistent.

Types, argument order, parameter names, etc. all play a role here.

Think about APIs you've used where you didn't need to look at documentation to use a new method.

Needle, Haystack vs. Haystack, Needle

```
# shell commands
cp from to
mv from to
ln from to
```

```
// PHP
strpos(string $haystack, string $needle)
in_array(mixed $needle, array $haystack)
```

Different Names, Same Thing

```
HttpRequest.header_dict
HttpRequest.body

HttpResponse.headers
HttpResponse.text
```

Wind up looking it up every time. This kind of thing also confuses LLMs if that's a concern.

Different Things, Same Name

```
web_ui.get_client() → WebBrowser()
accounts.get_client() → {"name": "Cody", "balance": 1000}
```

Confusing! Use consistent language throughout application even if synonyms are needed.

Set of Primitives

Define a clear set of data models and/or behaviors important to the domain.

Can then ensure consistency and ease mental burden

Sets of Primitives

- **CRUD**: Create, Read, Update, Delete
  - Consistency! createuser vs. adduser / createdb vs. initdb
  - We'll revisit consistency when we discuss REST APIs
- `load/dump`
- **GIS Primitives**: Point, Line, Polygon, intersect, overlap, etc.

```
intersect(Line, Line) → MultiPoint
intersect(Line, Polygon) → MultiLine
intersect(Polygon, Polygon) → MultiPolygon
```

Interface Design Patterns

Facade

Adapter

Strategy

Facade

Simplify complex/leaky interface with a thin layer that exposes only *necessary* options.

sending an email using python stdlib

```
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart

message = MIMEMultipart()
message["From"] = sender_email
message["To"] = recipient_email
message["Subject"] = "Test Email"

body = "This is a test email sent from Python."
message.attach(MIMEText(body, "plain"))

server = smtplib.SMTP(smtp_server, smtp_port)
server.starttls()
server.login(sender_email, sender_password)
server.send_message(message)
```

Django’s send_mail facade

```
def send_mail(
  subject,
  message,
  from_email,
  recipient_list,
  fail_silently=False,
  auth_user=None,
  auth_password=None,
  connection=None,
  html_message=None)
```

Reduces complexity of sending email down to 10 possible parameters.

(Still quite a lot, but underlying mail libraries have a lot more possible behavior.)

Adapter

Wrap given interface in expected interface.

Suppose you have an application w/ `save_table(data, io_module)`

It expects `io_module` to implement the `dump/load` interface, but `pyxl` does not!

```
# excel_adapter.py
def load(f: file) → list | dict:
     workbook = pyxl.load_workbook(f)
     data = []
     for sheet in workbook:
          ...
     return sheet_data

def dump(data: list | dict, f: file):
    workbook = pyxl.Workbook()
     ...
    workbook.save(f)
```

Strategy

Define a family of algorithms and make them interchangeable.

Functional: Pass in a strategy function

```
sorted(data, key=lambda v: v.lower())
```

The second parameter is used to extract a *sortable interface* from each item in `data`.

OOP: Define a strategy in subclass

```
# BillScraper defines common behavior, TexasBillScraper specializes

class TexasBillScraper(BillScraper):
    def extract_bill_sponsors(self, **bill_data):
      ...
```

Typed Python

Allows being more specific about interfaces.

```
def func(
  param: str,
  collection: list[int],
  optional: str | None,
  multi: int | float
) → ReturnType:
```

Document interfaces in human & machine-readable way.

Not Enforced by Python, use a *type-checker*: `mypy` or `ty` in your continuous integration to ensure interfaces are followed. Catch errors before runtime.

Force you to think through types! (Hard to type? often a problematic interface)
https://mypy.readthedocs.io/en/stable/cheat_sheet_py3.html

Dataclasses

```
from dataclasses import dataclass
from enum import Enum, auto

class StopType(Enum):
    BUS = auto()
    METRA = auto()
    CTA = auto()

@dataclass
class TransitStop:
    stop_id: str
    name: str
    latitude: float
    longitude: float
    is_wheelchair_accessible: bool = False
    stop_type: StopType = StopType.BUS
```

```
TransitStop(
  stop_id=123,
  name="Washington Park",
  latitude=41.791,
  longitude=-87.612,
  stop_type=StopType.CTA
)
```