Skip the Design Patterns: Architecting with Nouns and Verbs

Brandon Rhodes

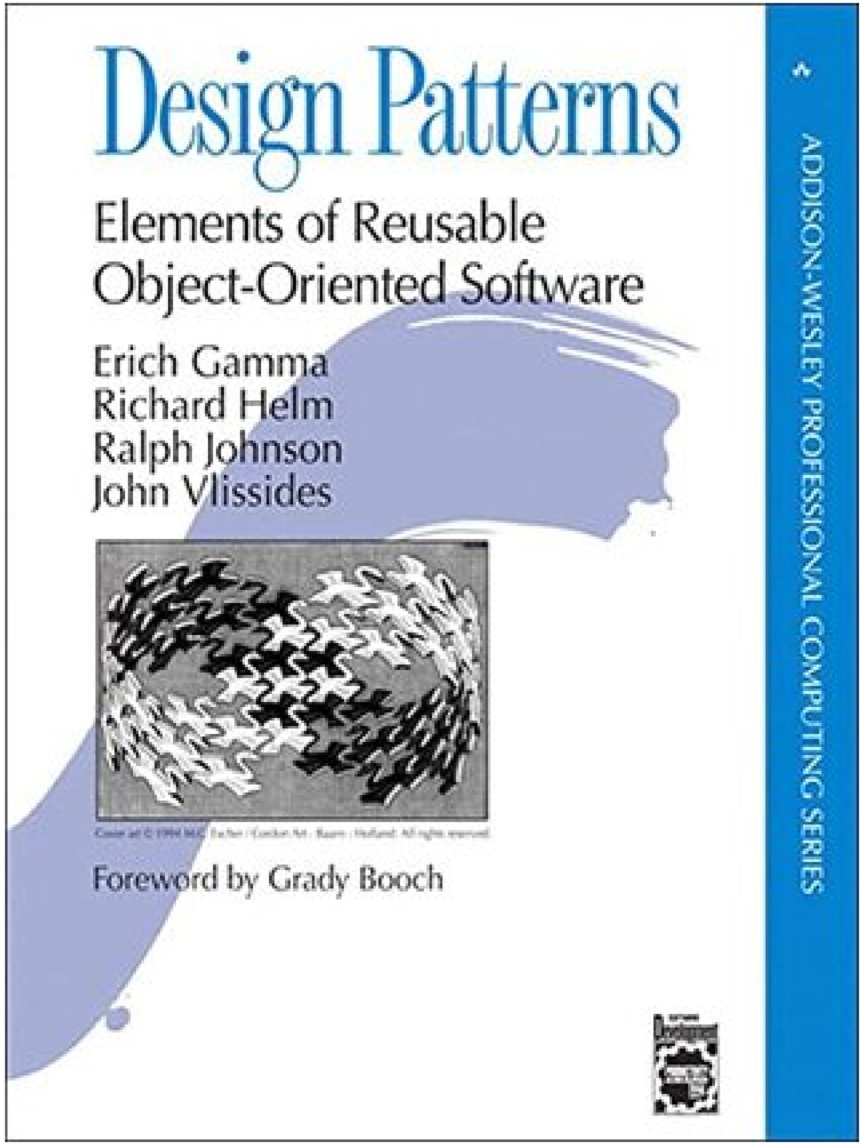
PyConLT 2025
Vilnius

Background:

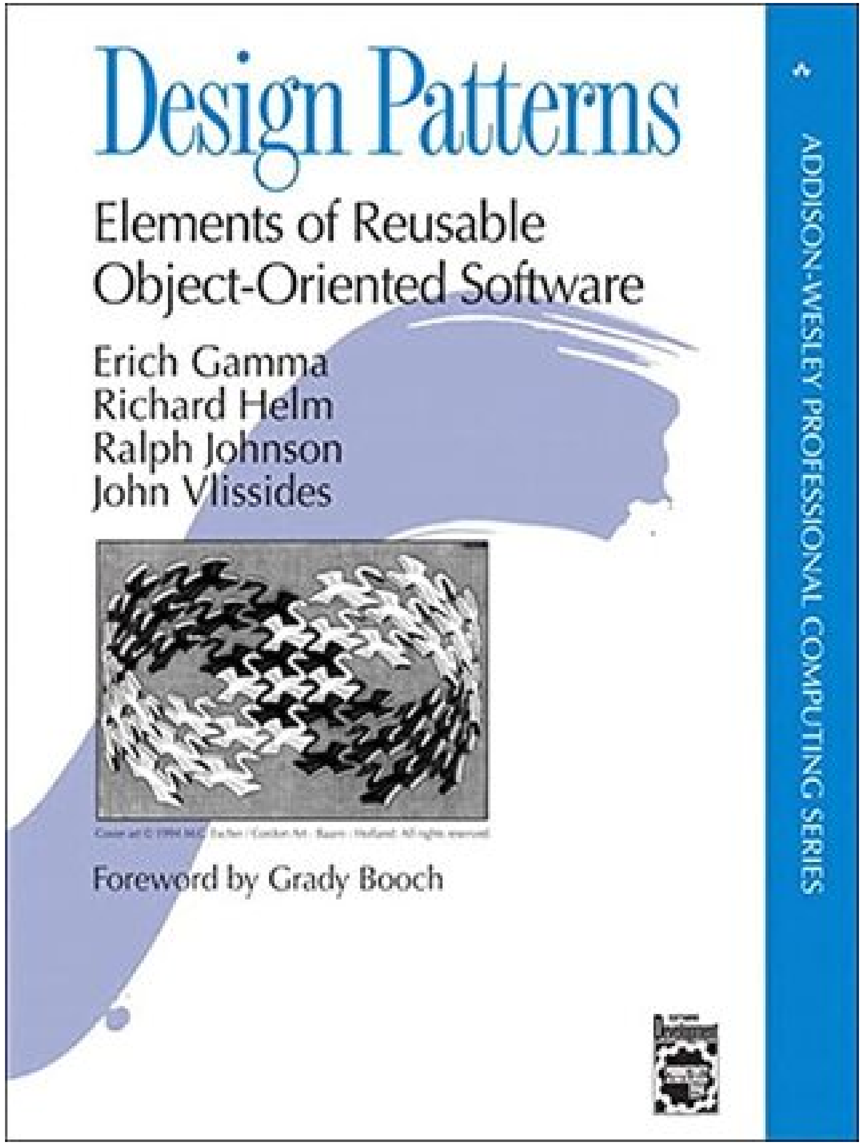
A decade ago, I noticed blogs that translated old ‘Design Patterns’ into Python.

Background:

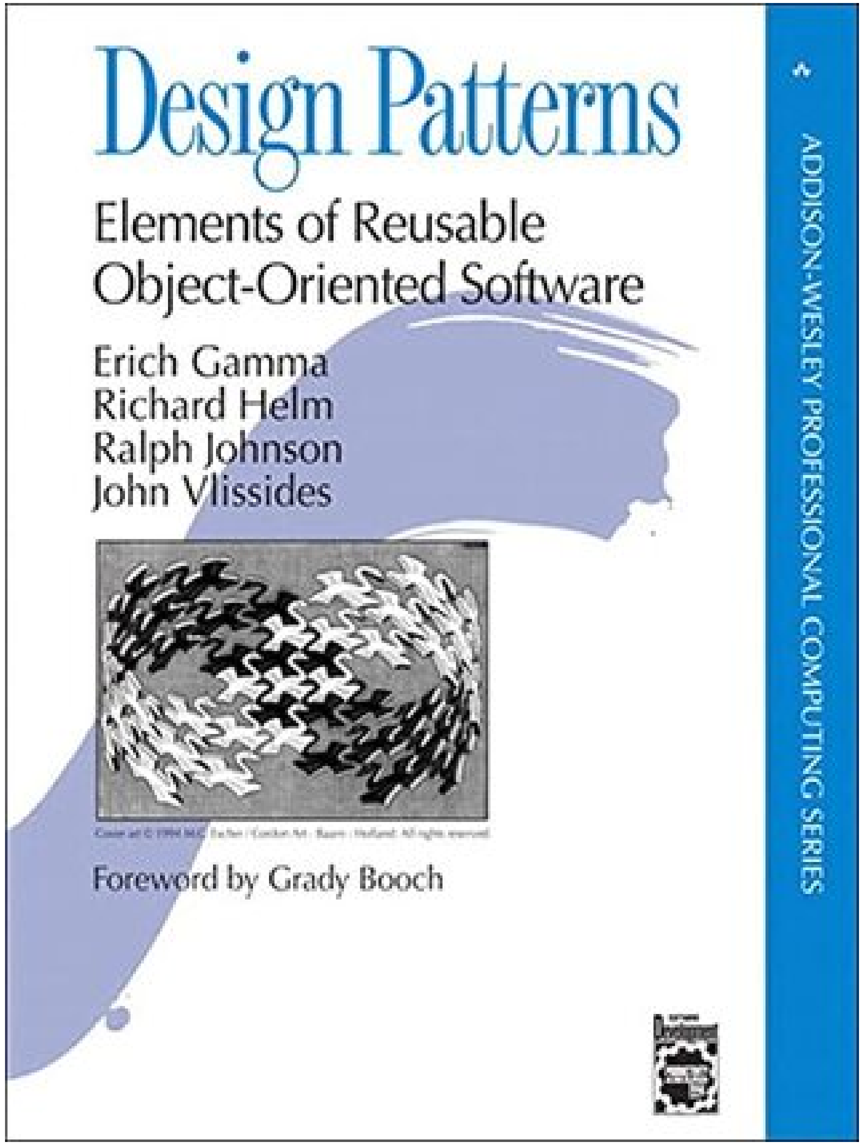
A decade ago, I noticed blogs that translated old 'Design Patterns' into Python.



I was annoyed.

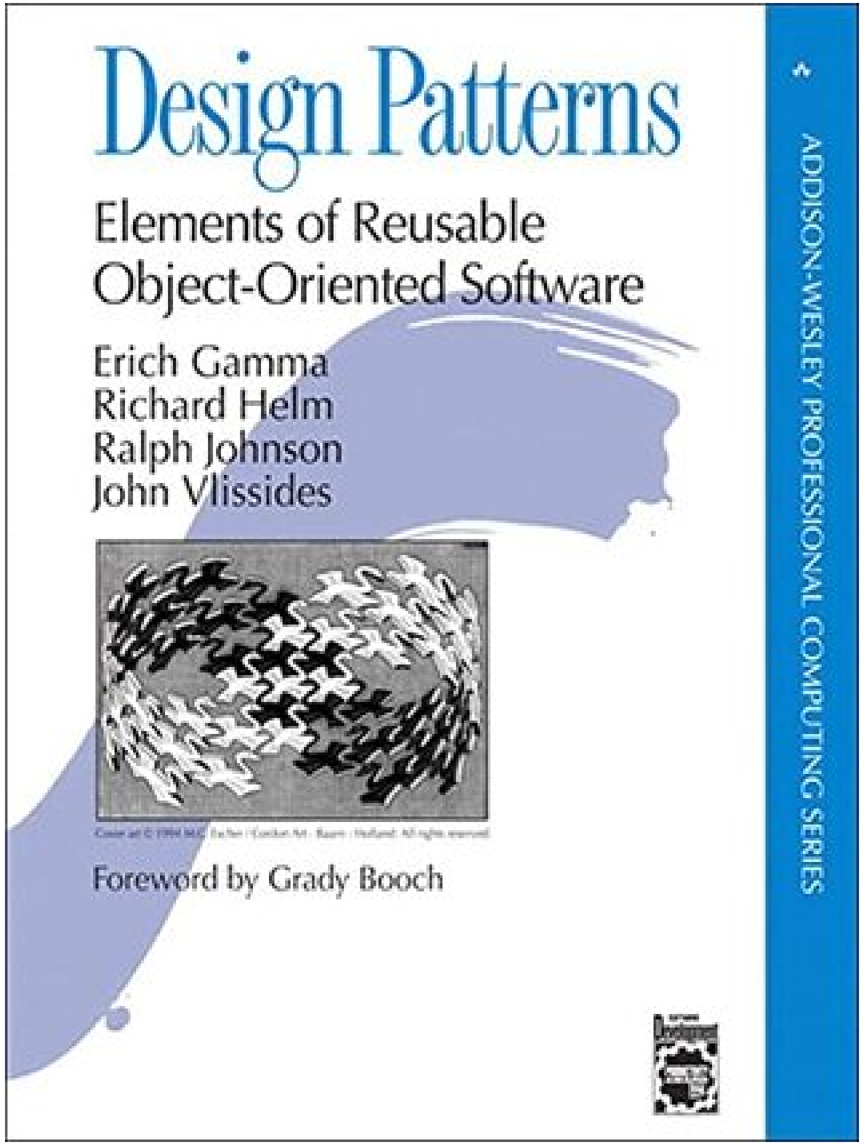


The blogs all translated the ‘Singleton’ design pattern into Python,



The blogs all translated the ‘Singleton’ design pattern into Python,

without mentioning that you shouldn’t use it!



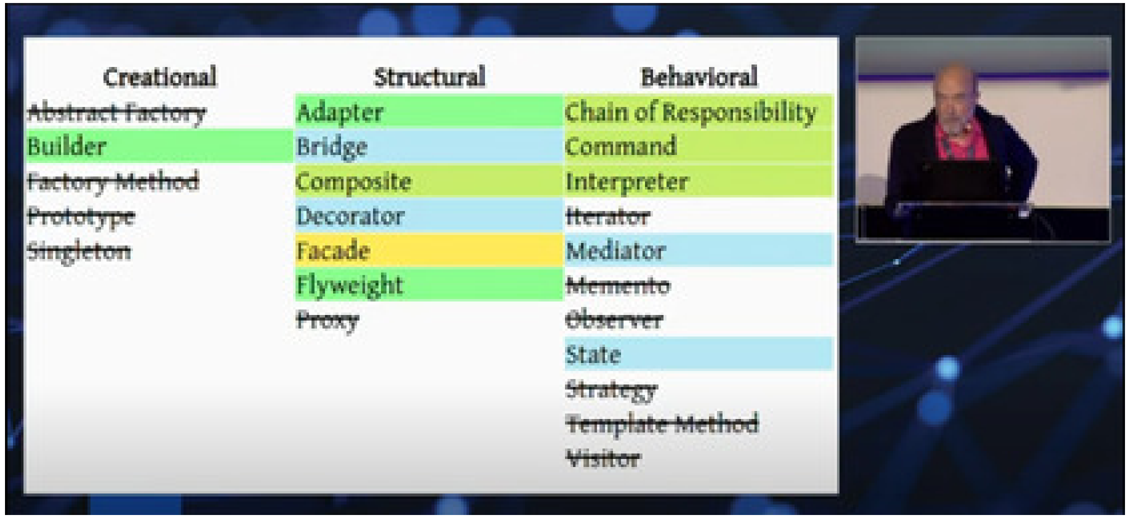
So I engaged in some
annoyance-driven
writing.

`https://python-patterns.guide`

And in some annoyance-driven talking.

2012 talk
‘Python Design Patterns’

2023 talk
‘The Classic Design Patterns: Where Are They Now?’



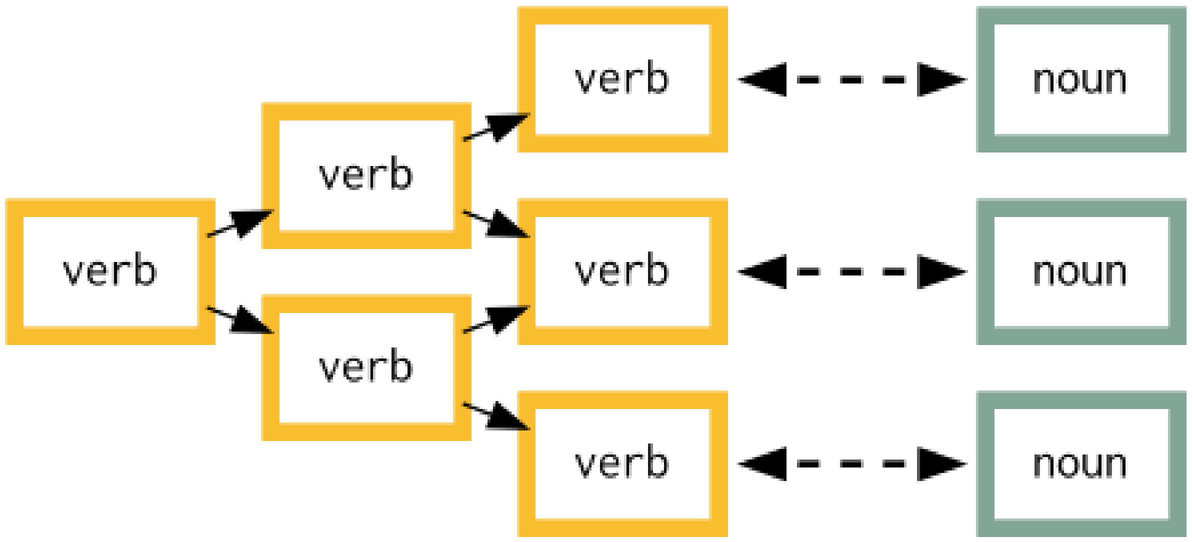
Big un-answered question:

If we don’t write Python using the old Design Patterns, what approach are we following instead?

I think modern Python code
is built from two kinds of object.

Verbs

stateless
idempotent
functions

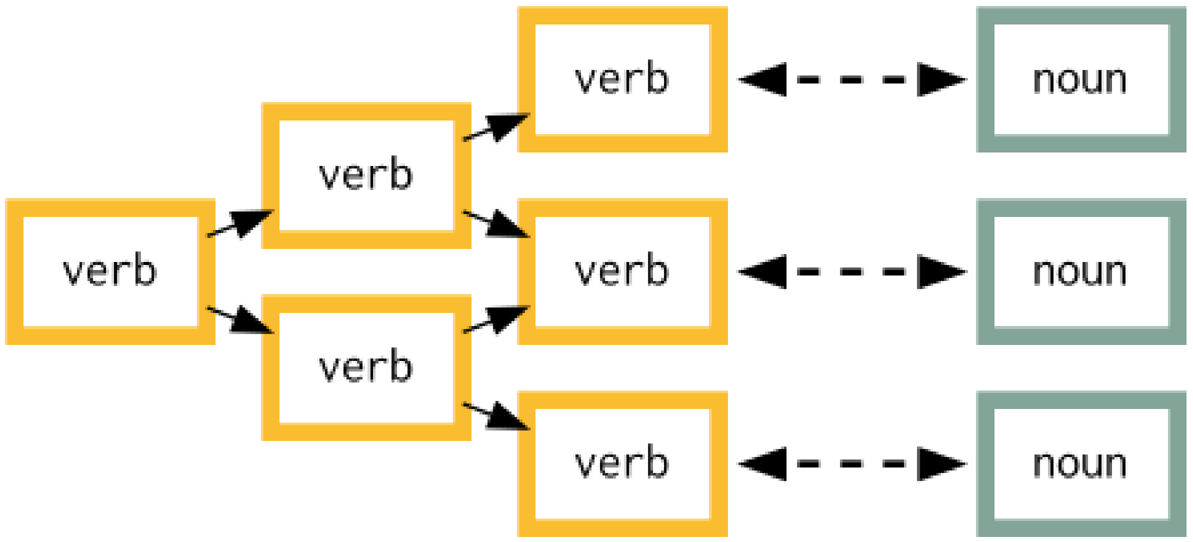


Verbs

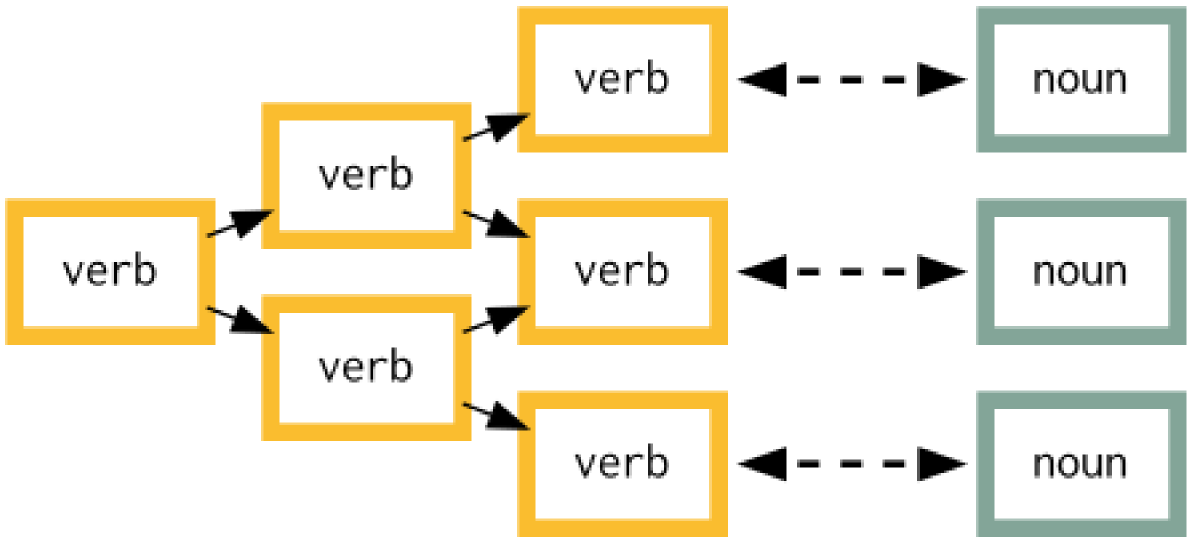
stateless
idempotent
functions

Nouns

stateful
mutable
data



- Verbs can stack and compose.
- Methods of nouns should be shallow



I think Python programmers learn
by progressing through
three stages.

1. Procedural
2. Abstraction
3. Encapsulation

1. Procedural ← All code in main()
2. Abstraction
3. Encapsulation

1. Procedural ← All code in main()
2. Abstraction ← Functions
3. Encapsulation

1. Procedural ← All code in main()
2. Abstraction ← Functions
3. Encapsulation ← Classes

1. Procedural	← Use nouns and verbs.
2. Abstraction	← Invent new verbs.
3. Encapsulation	← Invent new nouns.

1. Procedural ← Easy
2. Abstraction
3. Encapsulation ← Hard

1. Procedural ← Novice
2. Abstraction
3. Encapsulation ← Expert

1. Procedural
2. Abstraction
3. Encapsulation ← Object Orientation starts you HERE

Let’s write some sample code
to illustrate the first two levels:

1. Procedural
2. Abstraction

But first, a note on naming—

Singular

vs.

Plural

Old days, singular was my default:

```
timestamp
time
t
```

Old days, singular was my default:

`timestamp`
`time`
`t`

It was the plural that was special:

`time_array`
`time_list`
`time_seq`
`times`

But then I started
using NumPy, where

plural

is the default.

But then I started
using NumPy, where

plural

is the default.

```
from numpy import array, sqrt

def pythagoras(x, y):
  return sqrt(x*x + y*y)

# Is `x` a float?
# Or an array?

pythagoras(3.0, 4.0)

pythagoras(
  array([3, 6, 9]),
  array([4, 8, 12]),
)
```

NumPy gave me the habit of singular names for sequences:

```
x = [0, 1, 2, 3]
```

```
from numpy import array, sqrt

def pythagoras(x, y):
  return sqrt(x*x + y*y)

# Is `x` a float?
# Or an array?

pythagoras(3.0, 4.0)

pythagoras(
  array([3, 6, 9]),
  array([4, 8, 12]),
)
```

(See also:
relational databases,
a table is not named
'Users' but 'User')

```
from numpy import array, sqrt

def pythagoras(x, y):
  return sqrt(x*x + y*y)

# Is `x` a float?
# Or an array?

pythagoras(3.0, 4.0)

pythagoras(
  array([3, 6, 9]),
  array([4, 8, 12]),
)
```

If this is the plural —

`x`

— then what is the singular?

`x_item`
`x_member`
`x_element`

```
from numpy import array, sqrt

def pythagoras(x, y):
  return sqrt(x*x + y*y)

# Is `x` a float?
# Or an array?

pythagoras(3.0, 4.0)

pythagoras(
  array([3, 6, 9]),
  array([4, 8, 12]),
)
```

If this is the plural —

`x`

— then what is the singular?

`x_item`
`x_member`
`x_element`

$$t_0, t_1, t_2, \ldots, \ t_i, \ \ldots, t_n$$

Here's my convention:

time_i
temperature_i
humidity_i

Except, if it's just one letter, not
t_i
but:

ti

$$t_0, t_1, t_2, \ldots, \; t_i, \; \ldots, t_n$$

So in the code that follows
my loops will look like:

```
for ti in t:
```

So in the code that follows
my loops will look like:

```
for ti in t:
```

Also, I’ll call Python lists
‘arrays’ because, too much NumPy.

Goal: use ‘plotly’ to plot
the temperature and humidity
recorded on a Raspberry Pi.

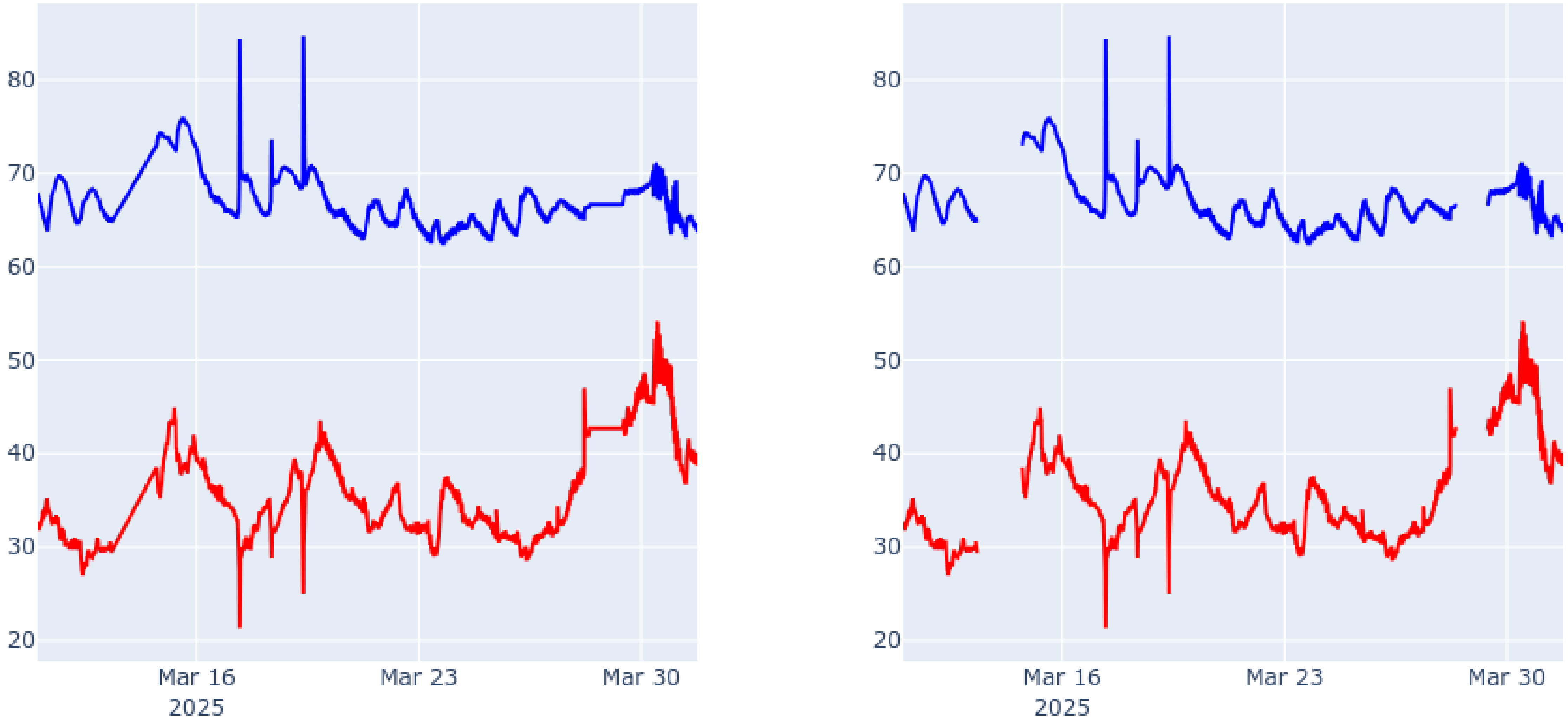
Goal: use ‘plotly’ to plot
the temperature and humidity
recorded on a Raspberry Pi.

```
# main()

t = [<datetime>, ...]
data = {'°C': [18.3, ...],
        '%H': [41.4, ...]}

fig.add(go.Scatter(
    x=t, y=data['°C'],
))
fig.add(go.Scatter(
    x=t, y=data['%H'],
))
```


```
# main()

t = [<datetime>, ...]
data = {'°C': [18.3, ...],
        '%H': [41.4, ...]}

fig.add(go.Scatter(
    x=t, y=data['°C'],
))
fig.add(go.Scatter(
    x=t, y=data['%H'],
))
```

Problem:

When the Raspberry Pi is unplugged, the plot has misleading straight lines.



80
70
60
50
40
30
20
Mar 16
2025
Mar 23
Mar 30

Here’s how the data
structures look.

```
t = [<datetime>, ...]
d = [18.3, 18.4, ...]
```

In a few places,
the time jumps by
several hours instead
of 1 minute.

```
t  [......../.../...........]
d  [........................]
```

Here’s how the data structures look.

```
t = [<datetime>, ...]
d = [18.3, 18.4, ...]
```

In a few places, the time jumps by several hours instead of 1 minute.

Here’s how the data structures look.

```
t = [<datetime>, ...]
d = [18.3, 18.4, ...]
```

In a few places, the time jumps by several hours instead of 1 minute.

```
t  [......./.../...........]
d  [.......................]
```

`I want 3 go.Scatter() calls:`

```
x  [........]
y  [........]

x          [....]
y          [....]

x               [...........]
y               [...........]
```


```
t  [......./.../...........]
d  [.......................]

 I want 3 go.Scatter() calls:

x  [........]
y  [........]

x          [....]
y          [....]

x              [...........]
y              [...........]
```

How will this
look in code?

Let’s first try it with
only one data series.

```
t  [......../.../............]
d  [.........................]

 I want 3 go.Scatter() calls:

x  [........]
y  [........]

x           [....]
y           [....]

x               [............]
y               [............]
```

How will this
look in code?

Let’s first try it with
only one data series.

```
# main()

t = [<datetime>, ...]
data = {'°C': [18.3, ...],
        '%H': [41.4, ...]}

fig.add(go.Scatter(
    x=t, y=data['°C'],
))
```

```
# main()

t = [<datetime>, ...]
data = {'°C': [18.3, ...],
        '%H': [41.4, ...]}

-fig.add(go.Scatter(
-    x=t, y=data['°C'],
-))
```

```
# main()

t = [<datetime>, ...]
data = {'°C': [18.3, ...],
        '%H': [41.4, ...]}

+segments = split(t, data['°C'])
+for x, y in segments:
+  fig.add(go.Scatter(x=x, y=y))
```

So, next, I tried writing split().

```
# main()

t = [<datetime>, ...]
data = {'°C': [18.3, ...],
        '%H': [41.4, ...]}

+segments = split(t, data['°C'])
+for x, y in segments:
+  fig.add(go.Scatter(x=x, y=y))
```

```
def split(t, d):
  ...
```

So, next, I tried writing split().

```
def split(t, d):
  ...
```

```
 def split(t, d):
   ...
+  for ti in t:
+    if (ti - tprev
+      > timedelta(minutes=1)):
       ...
   ...
```

Distraction!

```
 def split(t, d):
   ...
+  for ti in t:
+    if (ti - tprev
+        > timedelta(minutes=1)):
       ...
   ...
```

μ-pattern #1

Extract a constant

```
def split(t, d):
    ...
+   for ti in t:
+     if (ti - tprev
+         > timedelta(minutes=1)):
        ...
    ...
```

```diff
 def split(t, d):
   ...
   for ti in t:
-    if (ti - tprev
-      > timedelta(minutes=1)):
       ...
   ...
```

```diff
+minute = timedelta(minutes=1)
+
 def split(t, d):
   ...
   for ti in t:
+    if ti - tprev > minute:
       ...
   ...
```

Where does `tprev` come from?

```
+minute = timedelta(minutes=1)
+
 def split(t, d):
     ...
     for ti in t:
+        if ti - tprev > minute:
             ...
     ...
```

μ-pattern #2

Save the previous element

```
+minute = timedelta(minutes=1)
+
 def split(t, d):
     ...
     for ti in t:
+        if ti - tprev > minute:
             ...
     ...
```

```
minute = timedelta(minutes=1)

def split(t, d):
  ...
  for ti in t:
    if ti - tprev > minute:
      ...
  ...
```

```
minute = timedelta(minutes=1)

def split(t, d):
  ...
  for ti in t:
    if ti - tprev > minute:
      ...
+    tprev = ti
  ...
```

What about the first time through the loop?

```
 minute = timedelta(minutes=1)

 def split(t, d):
   ...
   for ti in t:
     if ti - tprev > minute:
       ...
+    tprev = ti
   ...
```

‘Sentinel Value’?

Anti-pattern!

```
minute = timedelta(minutes=1)

def split(t, d):
    ...
    for ti in t:
      if ti - tprev > minute:
        ...
+     tprev = ti
    ...
```

```
minute = timedelta(minutes=1)

def split(t, d):
  ...
  for ti in t:
-   if ti - tprev > minute:
      ...
    tprev = ti
  ...
```

```
minute = timedelta(minutes=1)

def split(t, d):
  ...
+ tprev = None
  for ti in t:
+   if (tprev is not None and
+       ti - tprev > minute):
      ...
    tprev = ti
  ...
```

Awful!

- Extra 'is not None' test
- Type becomes 'datetime | None'

```
 minute = timedelta(minutes=1)

 def split(t, d):
   ...
+  tprev = None
   for ti in t:
+    if (tprev is not None and
+        ti - tprev > minute):
       ...
     tprev = ti
   ...
```

µ-pattern #3

Safe initial value

```
 minute = timedelta(minutes=1)

 def split(t, d):
    ...
+   tprev = None
    for ti in t:
+     if (tprev is not None and
+         ti - tprev > minute):
        ...
      tprev = ti
    ...
```

```diff
 minute = timedelta(minutes=1)

 def split(t, d):
   ...
-  tprev = None
   for ti in t:
-    if (tprev is not None and
-        ti - tprev > minute):
       ...
     tprev = ti
   ...
```

```diff
 minute = timedelta(minutes=1)

 def split(t, d):
   ...
+  tprev = t[0]
   for ti in t:
+    if ti - tprev > minute:
       ...
     tprev = ti
   ...
```

Let’s now build the two lists ‘x’ and ‘y’.

```
minute = timedelta(minutes=1)

def split(t, d):
  ...
  tprev = t[0]
  for ti in t:
    if ti - tprev > minute:
      ...
    tprev = ti
  ...
```

```
minute = timedelta(minutes=1)

def split(t, d):
  ...
  tprev = t[0]
  for ti in t:
    if ti - tprev > minute:
      ...
    tprev = ti
  ...
```

```
 minute = timedelta(minutes=1)

 def split(t, d):
+  x, y = [], []
   tprev = t[0]
   for ti in t:
     if ti - tprev > minute:
       ...
+    x.append(ti)
+    y.append(di)
     tprev = ti
   ...
```

Finally,
let’s ‘yield’
our result.

```
minute = timedelta(minutes=1)

def split(t, d):
  x, y = [], []
  tprev = t[0]
  for ti in t:
    if ti - tprev > minute:
      ...
    x.append(ti)
    y.append(di)
    tprev = ti
  ...
```

```python
minute = timedelta(minutes=1)

def split(t, d):
  x, y = [], []
  tprev = t[0]
  for ti in t:
    if ti - tprev > minute:
      ...
    x.append(ti)
    y.append(di)
    tprev = ti
  ...
```

```diff
 minute = timedelta(minutes=1)

 def split(t, d):
   x, y = [], []
   tprev = t[0]
   for ti in t:
     if ti - tprev > minute:
+      yield x, y
+      x, y = [], []
     x.append(ti)
     y.append(di)
     tprev = ti
   ...
```

I know what you're thinking:
did Brandon miss something?

```
 minute = timedelta(minutes=1)

 def split(t, d):
   x, y = [], []
   tprev = t[0]
   for ti in t:
     if ti - tprev > minute:
+      yield x, y
+      x, y = [], []
     x.append(ti)
     y.append(di)
     tprev = ti
   ...
```

μ-pattern #4

'Grouping' needs two 'yields'

```
minute = timedelta(minutes=1)

def split(t, d):
  x, y = [], []
  tprev = t[0]
  for ti in t:
    if ti - tprev > minute:
      yield x, y
      x, y = [], []
    x.append(ti)
    y.append(di)
    tprev = ti
  ...
```

```
minute = timedelta(minutes=1)

def split(t, d):
  x, y = [], []
  tprev = t[0]
  for ti in t:
    if ti - tprev > minute:
      yield x, y
      x, y = [], []
    x.append(ti)
    y.append(di)
    tprev = ti
  ...
```

```
minute = timedelta(minutes=1)

def split(t, d):
  x, y = [], []
  tprev = t[0]
  for ti in t:
    if ti - tprev > minute:
      yield x, y
      x, y = [], []
    x.append(ti)
    y.append(di)
    tprev = ti
+ yield x, y
```

‘I’m done!’

```
minute = timedelta(minutes=1)

def split(t, d):
  x, y = [], []
  tprev = t[0]
  for ti in t:
    if ti - tprev > minute:
      yield x, y
      x, y = [], []
    x.append(ti)
    y.append(di)
    tprev = ti
  yield x, y
```

NameError:
'di' is not defined.

```
minute = timedelta(minutes=1)

def split(t, d):
  x, y = [], []
  tprev = t[0]
  for ti in t:
    if ti - tprev > minute:
      yield x, y
      x, y = [], []
    x.append(ti)
    y.append(di)
    tprev = ti
  yield x, y
```

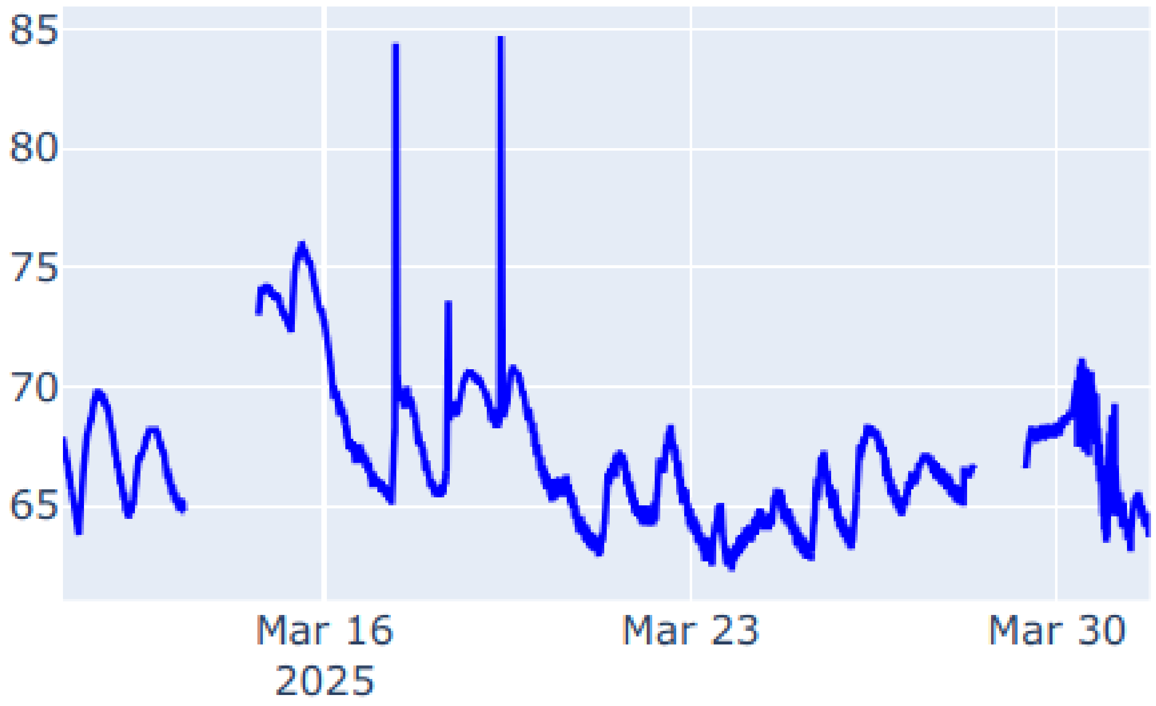
```
minute = timedelta(minutes=1)

def split(t, d):
  x, y = [], []
  tprev = t[0]
- for ti in t:
    if ti - tprev > minute:
      yield x, y
      x, y = [], []
    x.append(ti)
    y.append(di)
    tprev = ti
  yield x, y
```

```
minute = timedelta(minutes=1)

def split(t, d):
  x, y = [], []
  tprev = t[0]
+ for ti, di in zip(t, d):
    if ti - tprev > minute:
      yield x, y
      x, y = [], []
    x.append(ti)
    y.append(di)
    tprev = ti
  yield x, y
```

This time it worked!

```
minute = timedelta(minutes=1)

def split(t, d):
  x, y = [], []
  tprev = t[0]
  for ti, di in zip(t, d):
    if ti - tprev > minute:
      yield x, y
      x, y = [], []
    x.append(ti)
    y.append(di)
    tprev = ti
  yield x, y
```

This time it worked!



Only one more
goal: plot ‘%H’!

Only one more goal: plot ‘%H’!

```
# main()

t = [<datetime>, ...]
data = {'°C': [18.3, ...],
        '%H': [41.4, ...]}

segments = split(t, data['°C'])
for x, y in segments:
  fig.add(go.Scatter(x=x, y=y))
```

Only one more goal: plot ‘%H’!

```
# main()

t = [<datetime>, ...]
data = {'°C': [18.3, ...],
        '%H': [41.4, ...]}

segments = split(t, data['°C'])
for x, y in segments:
  fig.add(go.Scatter(x=x, y=y))

segments = split(t, data['%H'])
for x, y in segments:
  fig.add(go.Scatter(x=x, y=y))
```

And it worked —

```
# main()

t = [<datetime>, ...]
data = {'°C': [18.3, ...],
        '%H': [41.4, ...]}

segments = split(t, data['°C'])
for x, y in segments:
  fig.add(go.Scatter(x=x, y=y))

segments = split(t, data['%H'])
for x, y in segments:
  fig.add(go.Scatter(x=x, y=y))
```


```
# main()

t = [<datetime>, ...]
data = {'°C': [18.3, ...],
        '%H': [41.4, ...]}

segments = split(t, data['°C'])
for x, y in segments:
  fig.add(go.Scatter(x=x, y=y))

segments = split(t, data['%H'])
for x, y in segments:
  fig.add(go.Scatter(x=x, y=y))
```

But something seemed wrong.
So I read the code over again.

```
# main()

t = [<datetime>, ...]
data = {'°C': [18.3, ...],
        '%H': [41.4, ...]}

segments = split(t, data['°C'])
for x, y in segments:
  fig.add(go.Scatter(x=x, y=y))

segments = split(t, data['%H'])
for x, y in segments:
  fig.add(go.Scatter(x=x, y=y))
```

It’s doing duplicate work!

It’s finding all the gaps in ‘t’ twice!

```
# main()

t = [<datetime>, ...]
data = {'°C': [18.3, ...],
        '%H': [41.4, ...]}

segments = split(t, data['°C'])
for x, y in segments:
  fig.add(go.Scatter(x=x, y=y))

segments = split(t, data['%H'])
for x, y in segments:
  fig.add(go.Scatter(x=x, y=y))
```

μ-pattern #5

Avoid duplicate work

```
# main()

t = [<datetime>, ...]
data = {'°C': [18.3, ...],
        '%H': [41.4, ...]}

segments = split(t, data['°C'])
for x, y in segments:
  fig.add(go.Scatter(x=x, y=y))

segments = split(t, data['%H'])
for x, y in segments:
  fig.add(go.Scatter(x=x, y=y))
```

So —

can I expand `split()`
from splitting one list
to splitting many?

So —

can I expand `split()` from splitting one list to splitting many?

```
minute = timedelta(minutes=1)

def split(t, d):
  x, y = [], []
  tprev = t[0]
  for ti, di in zip(t, d):
    if ti - tprev > minute:
      yield x, y
      x, y = [], []
    x.append(ti)
    y.append(di)
    tprev = ti
  yield x, y
```

```
 minute = timedelta(minutes=1)

-def split(t, d):
-  x, y = [], []
   tprev = t[0]
-  for ti, di in zip(t, d):
     if ti - tprev > minute:
-      yield x, y
-      x, y = [], []
     x.append(ti)
-    y.append(di)
     tprev = ti
-  yield x, y
```

```
 minute = timedelta(minutes=1)

+def split(t, d_lists):
+  x, y… = [], [[],…]
   tprev = t[0]
+  for ti, di… in zip(t, d…):
     if ti - tprev > minute:
+      yield x, [y, …]
+      x, y… = [], [[],…]
     x.append(ti)
+    for yi, dii in zip(y, di…):
+      yi.append(dii)
     tprev = ti
+  yield x, [y, …]
```

NOPE

```
 minute = timedelta(minutes=1)

+def split(t, d_lists):
+  x, y… = [], [[],…]
   tprev = t[0]
+  for ti, di… in zip(t, d…):
     if ti - tprev > minute:
+      yield x, [y, …]
+      x, y… = [], [[],…]
     x.append(ti)
+    for yi, dii in zip(y, di…):
+      yi.append(dii)
     tprev = ti
+  yield x, [y, …]
```

The complexity
of this code is

outrunning

the complexity
of my problem.

```
 minute = timedelta(minutes=1)

+def split(t, d_lists):
+  x, y… = [], [[],…]
   tprev = t[0]
+  for ti, di… in zip(t, d…):
     if ti - tprev > minute:
+      yield x, [y, …]
+      x, y… = [], [[],…]
     x.append(ti)
+    for yi, dii in zip(y, di…):
+      yi.append(dii)
     tprev = ti
+  yield x, [y, …]
```

‘There must be a better way’

```
 minute = timedelta(minutes=1)

+def split(t, d_lists):
+  x, y… = [], [[],…]
   tprev = t[0]
+  for ti, di… in zip(t, d…):
     if ti - tprev > minute:
+      yield x, [y, …]
+      x, y… = [], [[],…]
     x.append(ti)
+    for yi, dii in zip(y, di…):
+      yi.append(dii)
     tprev = ti
+  yield x, [y, …]
```

μ-pattern #6

Do one thing at a time.

```
minute = timedelta(minutes=1)

def split(t, d):
  x, y = [], []
  tprev = t[0]
  for ti, di in zip(t, d):
    if ti - tprev > minute:
      yield x, y
      x, y = [], []
    x.append(ti)
    y.append(di)
    tprev = ti
  yield x, y
```

This code

both
tries to find the gaps
in the time series

and
split both lists
at those gaps.

```
minute = timedelta(minutes=1)

def split(t, d):
  x, y = [], []
  tprev = t[0]
  for ti, di in zip(t, d):
    if ti - tprev > minute:
      yield x, y
      x, y = [], []
    x.append(ti)
    y.append(di)
    tprev = ti
  yield x, y
```

Idea:

`gaps(t)`
to find the
index of each gap.

`split(...)`
that splits a list
at those indices.

Let’s try writing ‘gaps()’

```
minute = timedelta(minutes=1)

def split(t, d):
  x, y = [], []
  tprev = t[0]
  for ti, di in zip(t, d):
    if ti - tprev > minute:
      yield x, y
      x, y = [], []
    x.append(ti)
    y.append(di)
    tprev = ti
  yield x, y
```

```
minute = timedelta(minutes=1)

-def split(t, d):
   x, y = [], []
   tprev = t[0]
   for ti, di in zip(t, d):
      if ti - tprev > minute:
         yield x, y
         x, y = [], []
      x.append(ti)
      y.append(di)
      tprev = ti
   yield x, y
```

```
minute = timedelta(minutes=1)

+def gaps(t):
   x, y = [], []
   tprev = t[0]
   for ti, di in zip(t, d):
      if ti - tprev > minute:
         yield x, y
         x, y = [], []
      x.append(ti)
      y.append(di)
      tprev = ti
   yield x, y
```

```
 minute = timedelta(minutes=1)

 def gaps(t):
   x, y = [], []
   tprev = t[0]
-  for ti, di in zip(t, d):
     if ti - tprev > minute:
       yield x, y
       x, y = [], []
     x.append(ti)
     y.append(di)
     tprev = ti
   yield x, y
```

```
 minute = timedelta(minutes=1)

 def gaps(t):
   x, y = [], []
   tprev = t[0]
+  for ti in t:
     if ti - tprev > minute:
       yield x, y
       x, y = [], []
     x.append(ti)
     y.append(di)
     tprev = ti
   yield x, y
```

```
 minute = timedelta(minutes=1)

 def gaps(t):
   x, y = [], []
   tprev = t[0]
   for ti in t:
     if ti - tprev > minute:
-      yield x, y
       x, y = [], []
     x.append(ti)
     y.append(di)
     tprev = ti
-  yield x, y
```

```
 minute = timedelta(minutes=1)

 def gaps(t):
   x, y = [], []
   tprev = t[0]
   for ti in t:
     if ti - tprev > minute:
+      yield i
       x, y = [], []
     x.append(ti)
     y.append(di)
     tprev = ti
+  yield i
```

```diff
 minute = timedelta(minutes=1)

 def gaps(t):
-  x, y = [], []
   tprev = t[0]
   for ti in t:
     if ti - tprev > minute:
       yield i
-      x, y = [], []
-    x.append(ti)
-    y.append(di)
     tprev = ti
   yield i
```

```python
minute = timedelta(minutes=1)

def gaps(t):
  tprev = t[0]
  for ti in t:
    if ti - tprev > minute:
      yield i
    tprev = ti
  yield i
```

So much simpler!
Let’s try running it.

```
minute = timedelta(minutes=1)

def gaps(t):
  tprev = t[0]
  for ti in t:
    if ti - tprev > minute:
      yield i
    tprev = ti
  yield i
```

NameError:
'i' is not defined.

```
minute = timedelta(minutes=1)

def gaps(t):
  tprev = t[0]
  for ti in t:
    if ti - tprev > minute:
      yield i
    tprev = ti
  yield i
```

```
minute = timedelta(minutes=1)

def gaps(t):
  tprev = t[0]
- for ti in t:
    if ti - tprev > minute:
      yield i
    tprev = ti
  yield i
```

```
minute = timedelta(minutes=1)

def gaps(t):
  tprev = t[0]
+ for i, ti in enumerate(t):
    if ti - tprev > minute:
      yield i
    tprev = ti
  yield i
```

And it worked!

Let's return to main()
to think about how split()
is going to work.

```
minute = timedelta(minutes=1)

def gaps(t):
  tprev = t[0]
  for i, ti in enumerate(t):
    if ti - tprev > minute:
      yield i
    tprev = ti
  yield i
```

And it worked!

Let’s return to main()
to think about how split()
is going to work.

```
# main()

t = [<datetime>, ...]
data = {'°C': [18.3, ...],
        '%H': [41.4, ...]}

for i in gaps(t):
  x = ... t ...

  y = ... data['°C'] ...
  fig.add(go.Scatter(x=x, y=y))

  y = ... data['%H'] ...
  fig.add(go.Scatter(x=x, y=y))
```

It suddenly struck me,
'split()' already has a name.

```
# main()

t = [<datetime>, ...]
data = {'°C': [18.3, ...],
        '%H': [41.4, ...]}

for i in gaps(t):
  x = ... t ...

  y = ... data['°C'] ...
  fig.add(go.Scatter(x=x, y=y))

  y = ... data['%H'] ...
  fig.add(go.Scatter(x=x, y=y))
```

In fact,
it's built in to Python!

'slicing'

```
# main()

t = [<datetime>, ...]
data = {'°C': [18.3, ...],
        '%H': [41.4, ...]}

for i in gaps(t):
  x = ... t ...

  y = ... data['°C'] ...
  fig.add(go.Scatter(x=x, y=y))

  y = ... data['%H'] ...
  fig.add(go.Scatter(x=x, y=y))
```

In fact,
it's built in to Python!

'slicing'

```
# main()

t = [<datetime>, ...]
data = {'°C': [18.3, ...],
        '%H': [41.4, ...]}

for ... in gaps(t):
  x = t[i:j]

  y = data['°C'][i:j]
  fig.add(go.Scatter(x=x, y=y))

  y = data['%H'][i:j]
  fig.add(go.Scatter(x=x, y=y))
```

In fact,
it's built in to Python!

'slicing'

```
# main()

t = [<datetime>, ...]
data = {'°C': [18.3, ...],
        '%H': [41.4, ...]}

for i, j in gaps(t):
  x = t[i:j]

  y = data['°C'][i:j]
  fig.add(go.Scatter(x=x, y=y))

  y = data['%H'][i:j]
  fig.add(go.Scatter(x=x, y=y))
```

In fact,
it's built in to Python!

'slicing'

```
# main()

t = [<datetime>, ...]
data = {'°C': [18.3, ...],
        '%H': [41.4, ...]}

for i, j in segments(t):
  x = t[i:j]

  y = data['°C'][i:j]
  fig.add(go.Scatter(x=x, y=y))

  y = data['%H'][i:j]
  fig.add(go.Scatter(x=x, y=y))
```

The lesson here:
'Do one thing at a time'

By entangling gap-finding with list-building in split(), I wound up re-implementing *a Python built-in!*

```
# main()

t = [<datetime>, ...]
data = {'°C': [18.3, ...],
        '%H': [41.4, ...]}

for i, j in segments(t):
  x = t[i:j]

  y = data['°C'][i:j]
  fig.add(go.Scatter(x=x, y=y))

  y = data['%H'][i:j]
  fig.add(go.Scatter(x=x, y=y))
```

Now we just need
to tweak ‘gaps()’ to
turn it into ‘segments()’.

```
minute = timedelta(minutes=1)

def gaps(t):
  tprev = t[0]
  for i, ti in enumerate(t):
    if ti - tprev > minute:
      yield i
    tprev = ti
  yield i
```

```
 minute = timedelta(minutes=1)

-def gaps(t):
   tprev = t[0]
   for i, ti in enumerate(t):
     if ti - tprev > minute:
       yield i
     tprev = ti
   yield i
```

```
 minute = timedelta(minutes=1)

+def segments(t):
   tprev = t[0]
   for i, ti in enumerate(t):
     if ti - tprev > minute:
       yield i
     tprev = ti
   yield i
```

```
 minute = timedelta(minutes=1)

 def segments(t):
   tprev = t[0]
   for i, ti in enumerate(t):
     if ti - tprev > minute:
-      yield i
     tprev = ti
-  yield i
```

```
 minute = timedelta(minutes=1)

 def segments(t):
   tprev = t[0]
   for i, ti in enumerate(t):
     if ti - tprev > minute:
+      yield start, i
     tprev = ti
+  yield start, i
```

```
minute = timedelta(minutes=1)

def segments(t):
  tprev = t[0]
  for i, ti in enumerate(t):
    if ti - tprev > minute:
      yield start, i
    tprev = ti
  yield start, i
```

```
minute = timedelta(minutes=1)

def segments(t):
  tprev = t[0]
+ start = 0
  for i, ti in enumerate(t):
    if ti - tprev > minute:
      yield start, i
+     start = i
    tprev = ti
  yield start, i
```

Done!

I got my graph
with gaps in it—

```
minute = timedelta(minutes=1)

def segments(t):
  tprev = t[0]
  start = 0
  for i, ti in enumerate(t):
    if ti - tprev > minute:
      yield start, i
      start = i
    tprev = ti
  yield start, i
```


80
70
60
50
40
30
20
Mar 16
2025
Mar 23
Mar 30

—while only scanning ‘t’
once to find its gaps.

```
# main()

t = [<datetime>, ...]
data = {'°C': [18.3, ...],
        '%H': [41.4, ...]}

for i, j in segments(t):
  x = t[i:j]

  y = data['°C'][i:j]
  fig.add(go.Scatter(x=x, y=y))

  y = data['%H'][i:j]
  fig.add(go.Scatter(x=x, y=y))
```

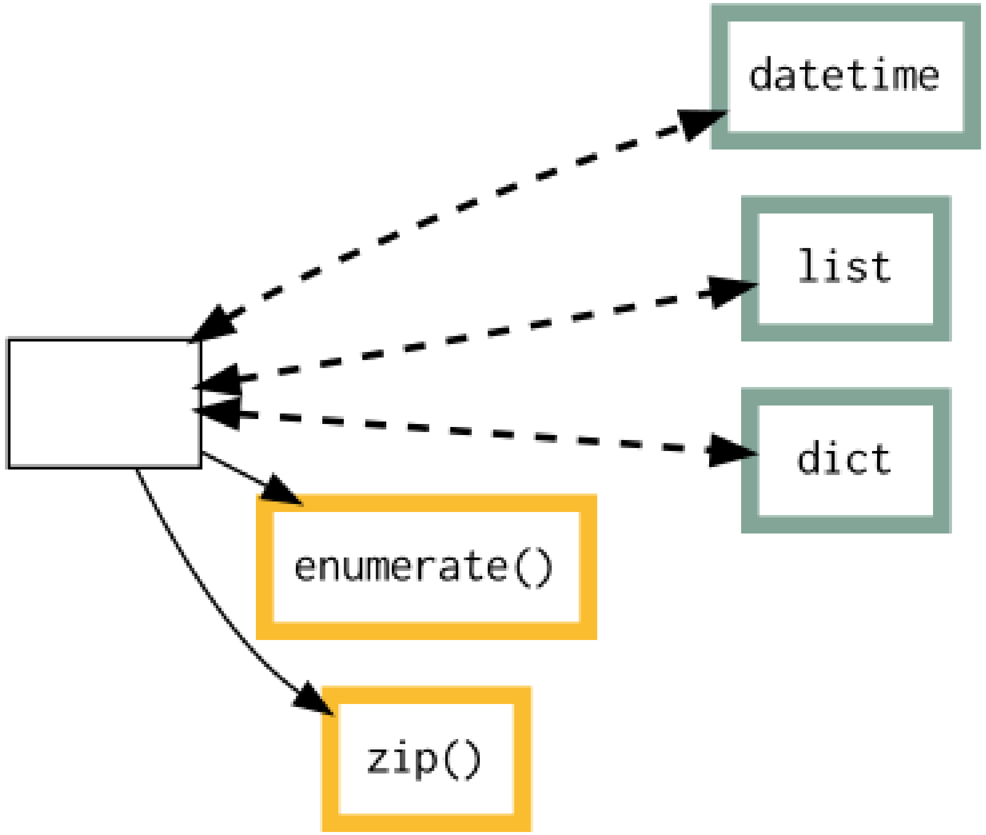
We are now finished writing example code. What levels was I operating at?

```
# main()

t = [<datetime>, ...]
data = {'°C': [18.3, ...],
        '%H': [41.4, ...]}

for i, j in segments(t):
  x = t[i:j]

  y = data['°C'][i:j]
  fig.add(go.Scatter(x=x, y=y))

  y = data['%H'][i:j]
  fig.add(go.Scatter(x=x, y=y))
```

1. Procedural
2. Abstraction
3. Encapsulation

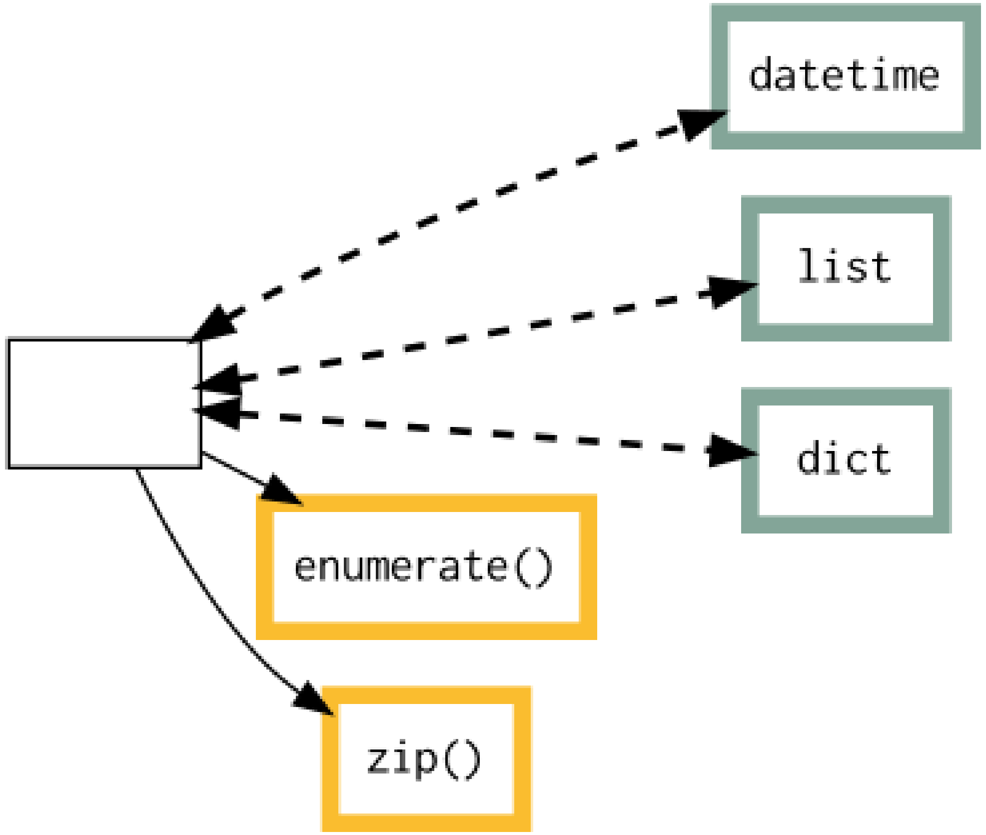
← Using nouns and verbs.

1. Procedural
2. Abstraction
3. Encapsulation



‘The Procedural Coder’

Gets to
use nouns and verbs
without having to design them.



‘The Procedural Coder’

Uses small ‘µ-patterns’ to arrange familiar lines of code to solve a new problem.

But I also operated
at a second level.

1. Procedural
2. Abstraction ← Invent new verbs.
3. Encapsulation

I wrote a
subroutine of my own.

`split() → gaps() → segments()`

What decisions do we face
when doing Abstraction?

Let’s look at three ‘Habits’
that I was following.

Why did I even introduce
a subroutine ‘segments()’?

Why didn’t I loop over
the list of datetimes
right here in main()?

```
# main()

t = [<datetime>, ...]
data = {'°C': [18.3, ...],
        '%H': [41.4, ...]}

for i, j in segments(t):
  x = t[i:j]

  y = data['°C'][i:j]
  fig.add(go.Scatter(x=x, y=y))

  y = data['%H'][i:j]
  fig.add(go.Scatter(x=x, y=y))
```

‘Brandon, you couldn’t!

```
# main()

t = [<datetime>, ...]
data = {'°C': [18.3, ...],
        '%H': [41.4, ...]}

for i, j in segments(t):
  x = t[i:j]

  y = data['°C'][i:j]
  fig.add(go.Scatter(x=x, y=y))

  y = data['%H'][i:j]
  fig.add(go.Scatter(x=x, y=y))
```

‘Brandon, you couldn’t!
It wouldn’t fit
on the slide.’

```
# main()

t = [<datetime>, ...]
data = {'°C': [18.3, ...],
        '%H': [41.4, ...]}

for i, j in segments(t):
  x = t[i:j]

  y = data['°C'][i:j]
  fig.add(go.Scatter(x=x, y=y))

  y = data['%H'][i:j]
  fig.add(go.Scatter(x=x, y=y))
```

Habit #1

'Don't mix levels of abstraction.'

```
# main()

t = [<datetime>, ...]
data = {'°C': [18.3, ...],
        '%H': [41.4, ...]}

for i, j in segments(t):
  x = t[i:j]

  y = data['°C'][i:j]
  fig.add(go.Scatter(x=x, y=y))

  y = data['%H'][i:j]
  fig.add(go.Scatter(x=x, y=y))
```

I/O just
feels like a

different level

than the little details
of iterating over a list.

```
# main()

t = [<datetime>, ...]
data = {'°C': [18.3, ...],
        '%H': [41.4, ...]}

for i, j in segments(t):
  x = t[i:j]

  y = data['°C'][i:j]
  fig.add(go.Scatter(x=x, y=y))

  y = data['%H'][i:j]
  fig.add(go.Scatter(x=x, y=y))
```

But, you might ask:

If I’m going
to add a subroutine,
why hide the ‘for’ loop?

Why not hide the I/O?

```
# main()

t = [<datetime>, ...]
data = {'°C': [18.3, ...],
        '%H': [41.4, ...]}

for i, j in segments(t):
  x = t[i:j]

  y = data['°C'][i:j]
  fig.add(go.Scatter(x=x, y=y))

  y = data['%H'][i:j]
  fig.add(go.Scatter(x=x, y=y))
```

Habit #2

Keep I/O near the top rather than burying it.

```
# main()

t = [<datetime>, ...]
data = {'°C': [18.3, ...],
        '%H': [41.4, ...]}

for i, j in segments(t):
  x = t[i:j]

  y = data['°C'][i:j]
  fig.add(go.Scatter(x=x, y=y))

  y = data['%H'][i:j]
  fig.add(go.Scatter(x=x, y=y))
```

Q: What happens when we bury I/O?

```
a()
  → b()
      → c()
          → go.Scatter()
```

A: The I/O gets hidden,
but not 'decoupled'.

```
a()
  → b()
      → c()
          → go.Scatter()
```

‘Tightly Coupled’

You can’t call a()
without it calling b(),
without that calling c() —

which calls go.Scatter().

```
a()
  → b()
      → c()
          → go.Scatter()
```

Symptom:

You try writing a test
of the code inside of ‘a()’,
and it saves a Plotly ‘.png’ to disk.

`mock.patch(go.Scatter)`
`mock.patch(c)`

```
a()
  → b()
      → c()
          → go.Scatter()
```

Cosmic Python!

www.cosmicpython.com

Harry J.W. Percival

Bob Gregory

```
a()
  → b()
      → c()
          → go.Scatter()
```

My approach, where possible,
is to keep I/O calls up
near the top.

‘Rebalancing’

```
a()
  → b()
      → c()
          → go.Scatter()
```

My approach, where possible,
is to keep I/O calls up
near the top.

‘Rebalancing’

```
main()
  → a()
    → b()
  → c()
  → go.Scatter()
```

‘Clean’ or ‘Hexagonal’ architecture.

```
main()
  → a()
    → b()
  → c()
  → go.Scatter()
```

But shallow call graphs have benefits even without I/O!

```
main()
  → a()
    → b()
  → c()
  → go.Scatter()
```

Habit #3

Shallow call graphs

testable
reusable

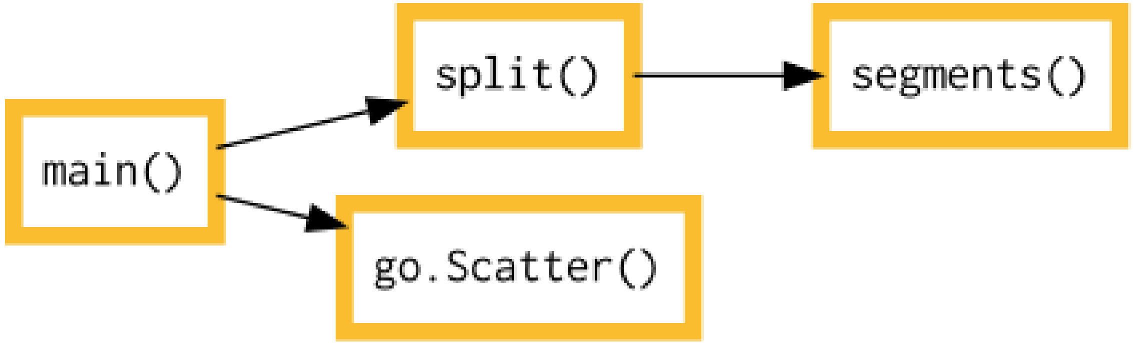
```
main()
  → a()
    → b()
  → c()
  → go.Scatter()
```

Subroutines
which are kept
close

are thereby kept
under control.

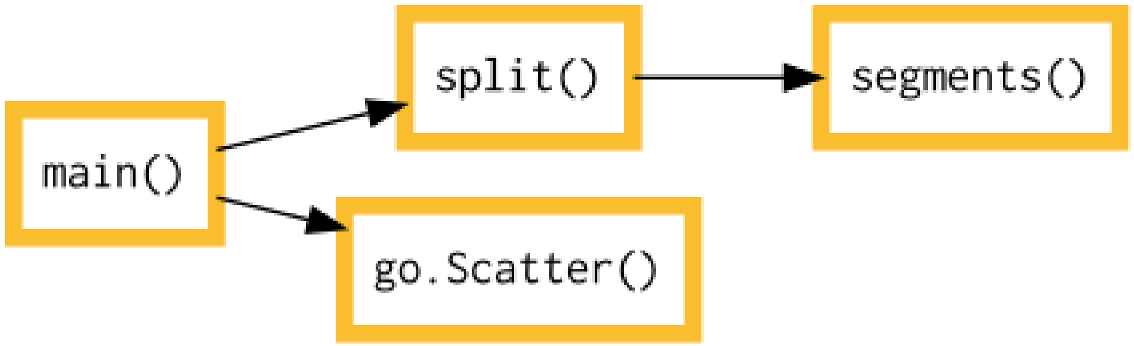
```
main()
  → a()
    → b()
  → c()
  → go.Scatter()
```

Imagine that I had built
a deep call graph.



‘New requirement!

‘We need to let
the user specify at
runtime how big a gap
“segments()” will detect.’



‘New requirement!

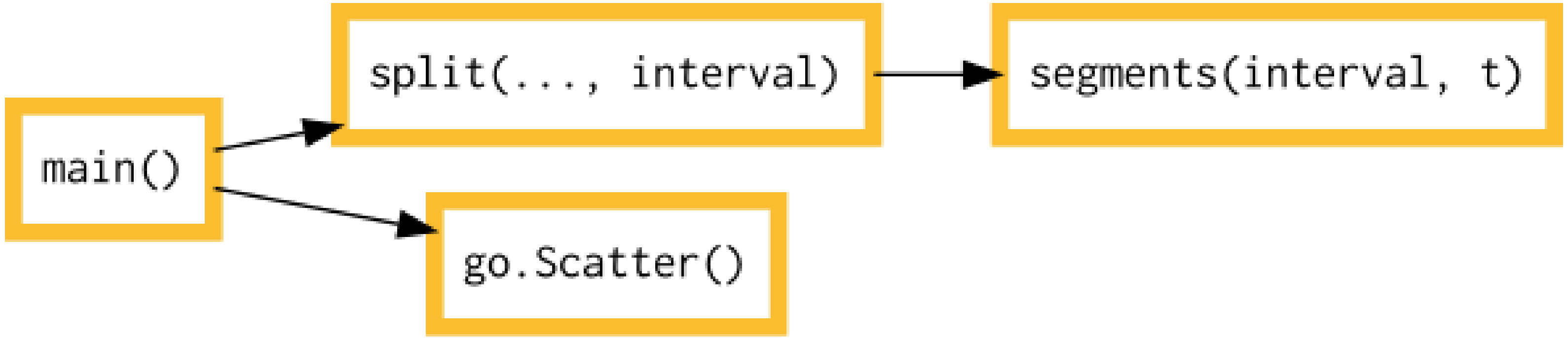
‘We need to let
the user specify at
runtime how big a gap
“segments()” will detect.’

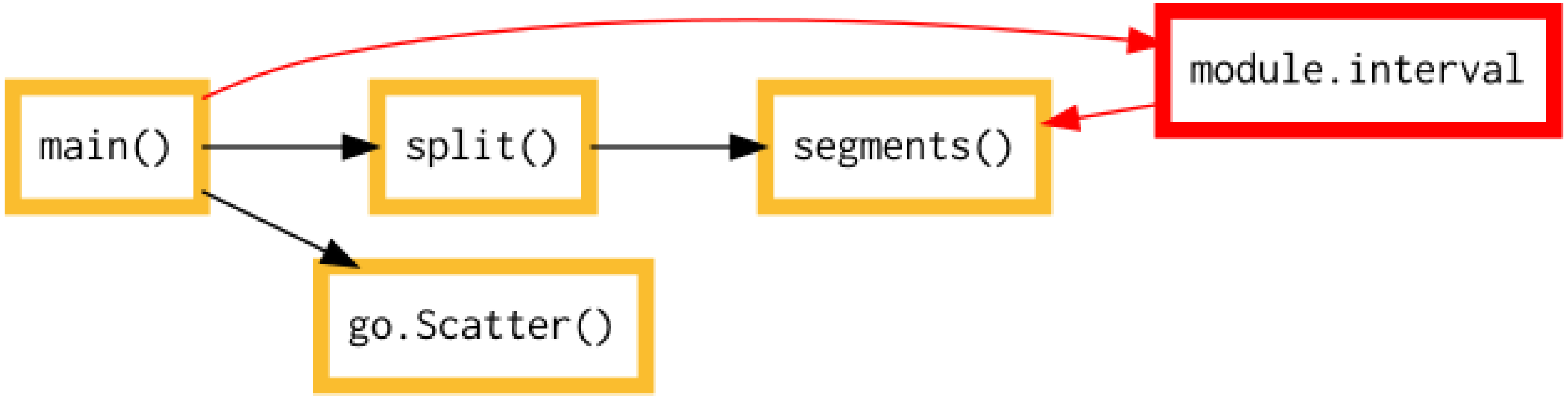
```
def segments(interval, t):
  ...
    if ti - tprev > interval:
  ...
```

In a deep call graph,
we have two options:

Pass ‘interval’ all the way down.
Store the ‘interval’ in a global.

```
def segments(interval, t):
  ...
    if ti - tprev > interval:
  ...
```


main()
split(..., interval)
segments(interval, t)
go.Scatter()


main()
split()
segments()
module.interval
go.Scatter()

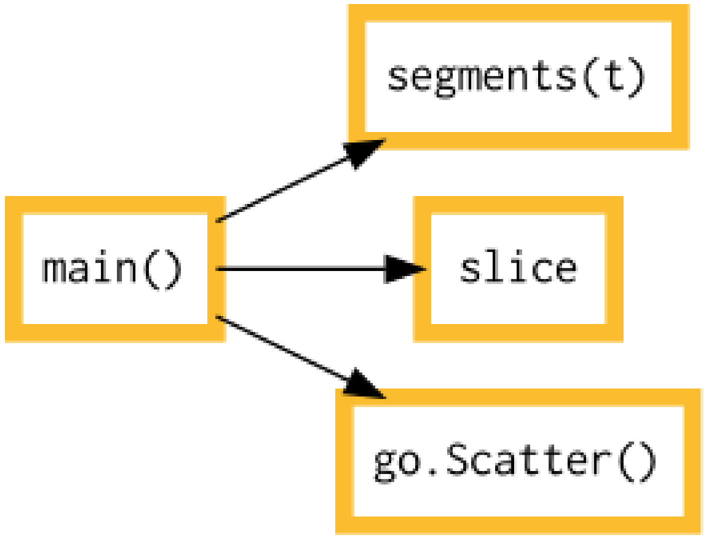
‘Scary action at a distance.’

Either way, main() can only
give the ‘interval’ to segments()
through a level of indirection.

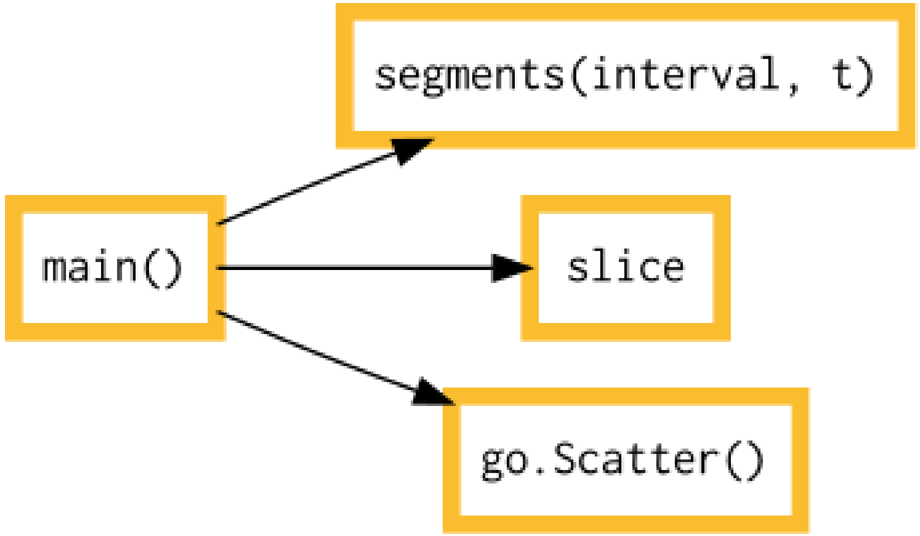
↓

Cost.

But if I can keep
my call graph shallow,
the problem is avoided.



But if I can keep
my call graph shallow,
the problem is avoided.



Abstraction

Don’t mix levels.
Decouple from I/O.
Shallow call graphs.

1. Procedural ← Use nouns and verbs.
2. Abstraction ← Invent new verbs.
3. Encapsulation

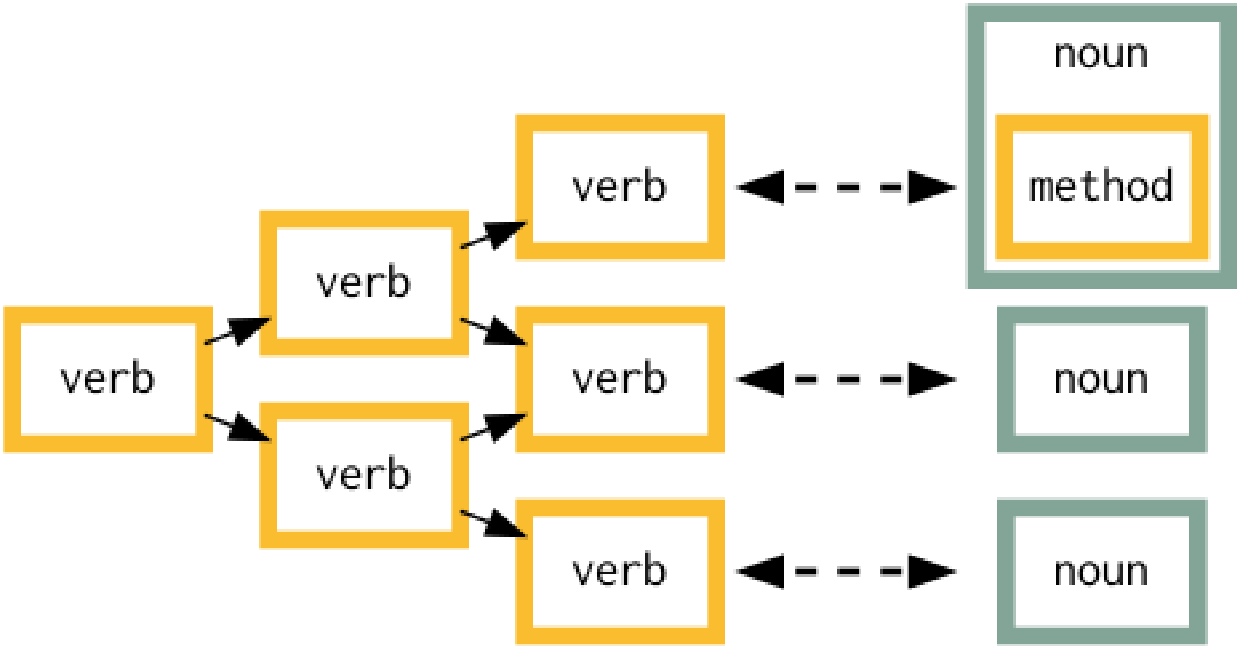
1. Procedural
2. Abstraction
3. Encapsulation ← Invent new nouns.

When have we embarked
upon ‘Encapsulation’?

When we attach
behavior
to
data.

When have we embarked upon ‘Encapsulation’?

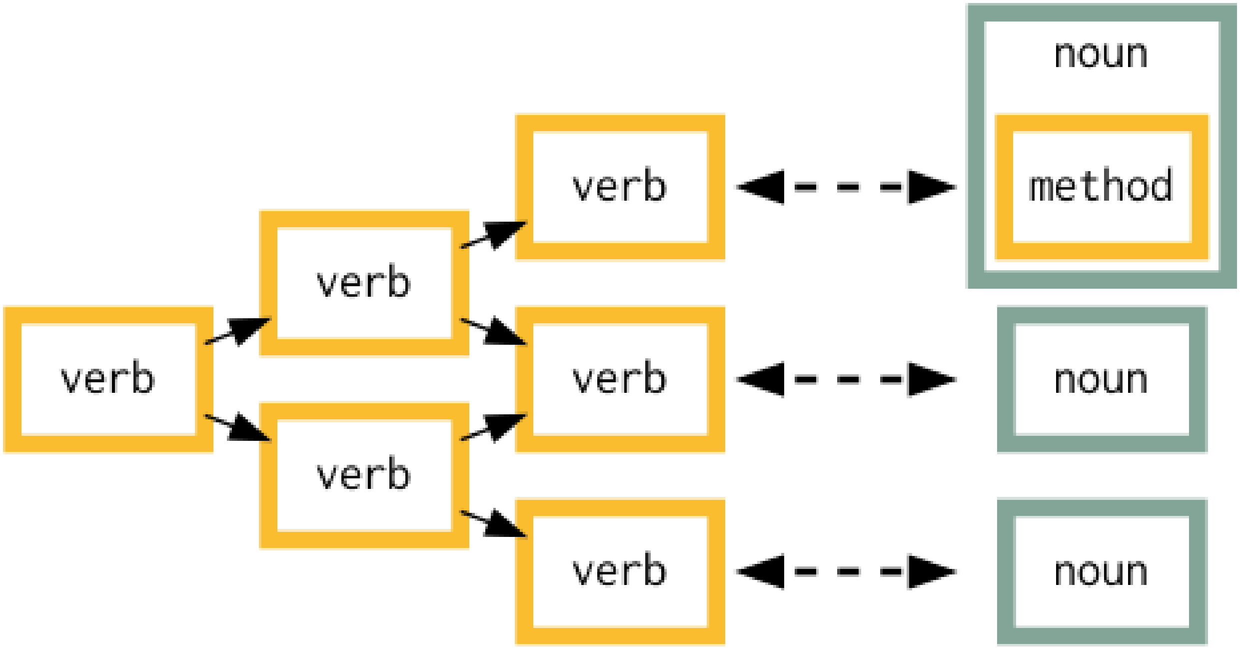
When we attach *behavior* to *data.*



Merely typing

`class`

doesn't mean you're doing Encapsulation!



Merely typing

`class`

doesn't mean you're doing Encapsulation!

```
# logging.py

...

class Filter:
  def __init__(self, pattern):
    self.pattern = pattern

  def call(self, message):
    b = self.pattern in message
    return b
```

Is this a noun?

```
# logging.py

...

class Filter:
  def __init__(self, pattern):
    self.pattern = pattern

  def call(self, message):
    b = self.pattern in message
    return b
```

This is a pure verb.

```
# logging.py

...

class Filter:
  def __init__(self, pattern):
    self.pattern = pattern

  def call(self, message):
    b = self.pattern in message
    return b
```

Verb

stateless
idempotent
function

```
# logging.py

...

class Filter:
  def __init__(self, pattern):
    self.pattern = pattern

  def call(self, message):
    b = self.pattern in message
    return b
```

This class implements
a verb that's customized
at runtime.

```
f = Filter('pattern').call
```

```
# logging.py

...

class Filter:
  def __init__(self, pattern):
    self.pattern = pattern

  def call(self, message):
    b = self.pattern in message
    return b
```

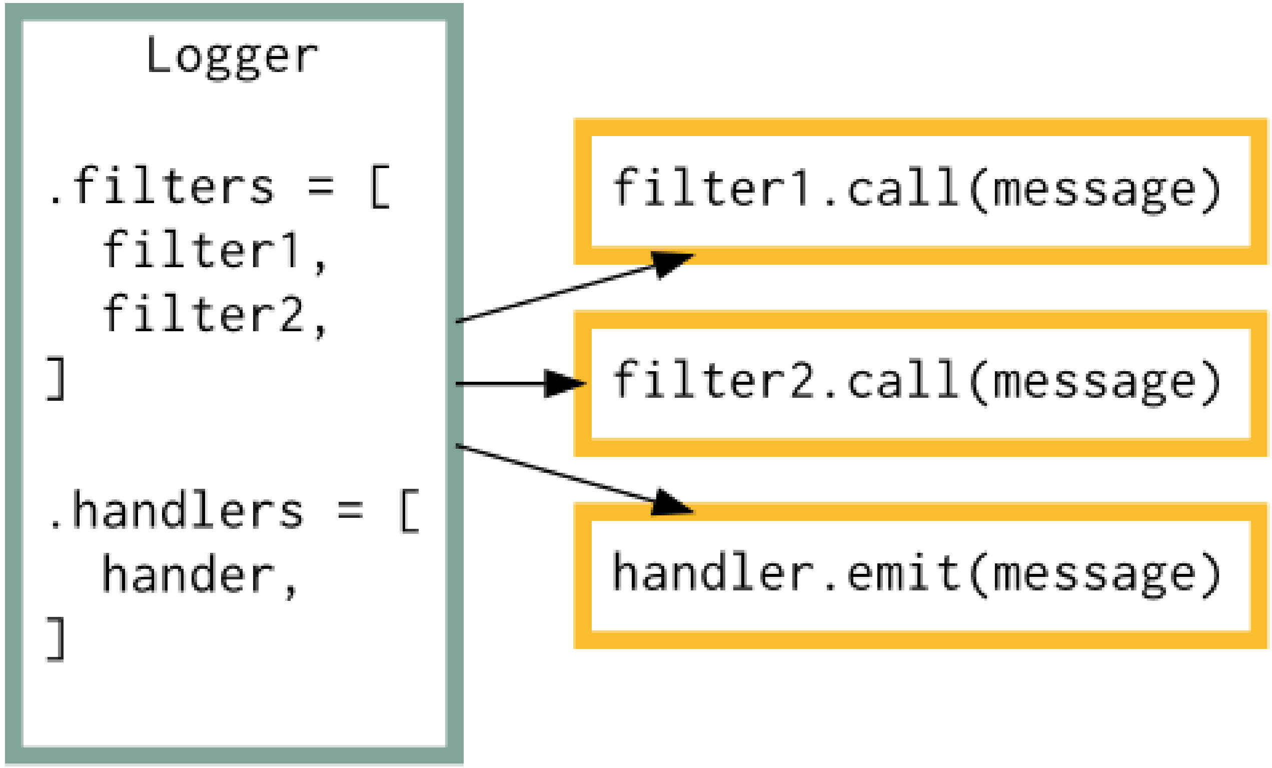
Why?

Because ‘logging’ often doesn’t learn its filters and handlers until runtime:

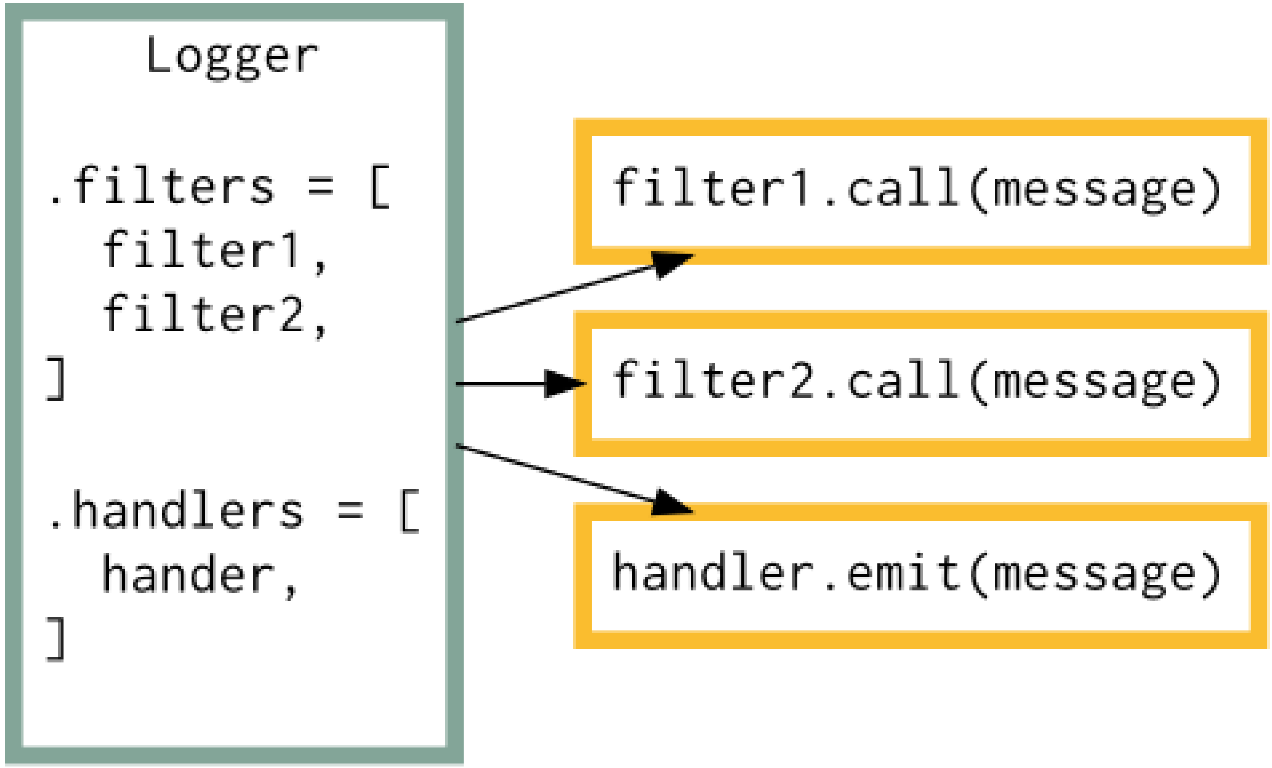
Command line options
Config.ini file

```
# logging.py

...

class Filter:
  def __init__(self, pattern):
    self.pattern = pattern

  def call(self, message):
    b = self.pattern in message
    return b
```

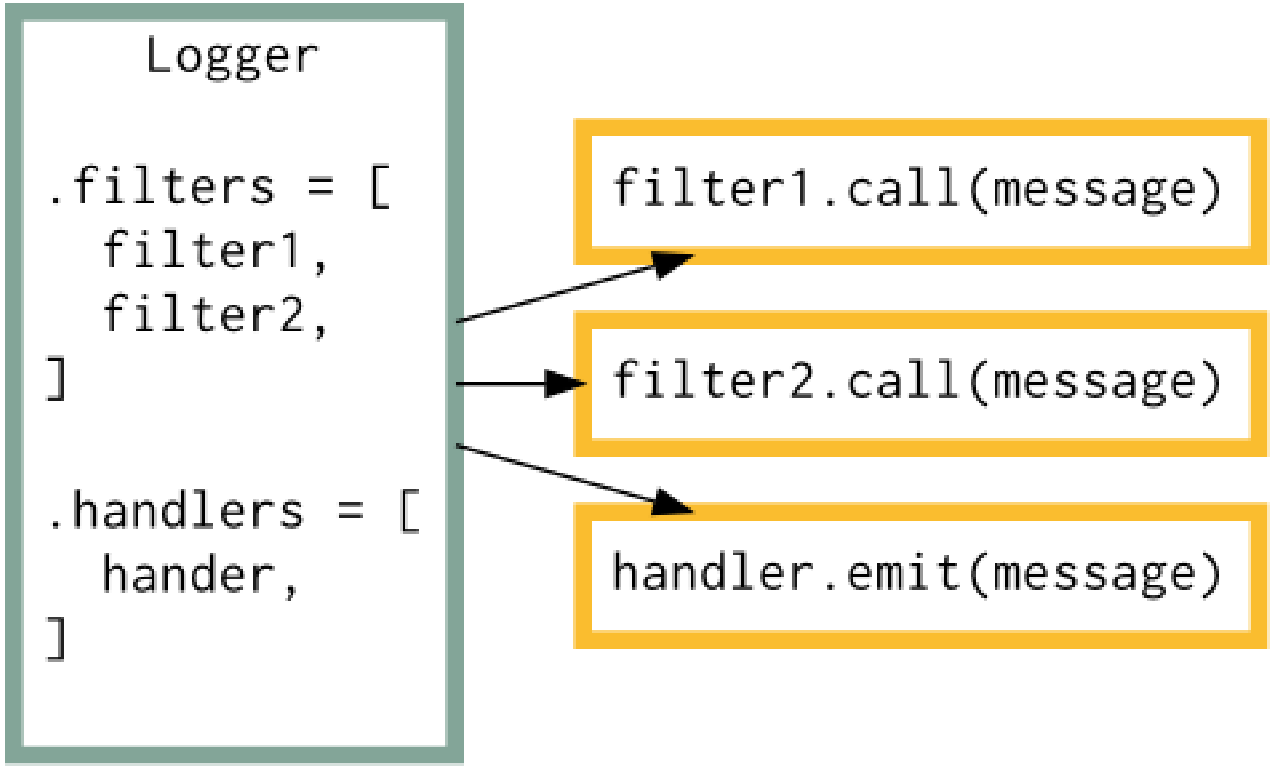
A configured logger keeps its 'Filter' objects and 'Handler' objects in plain Python lists.



When given a message,
the Logger calls its filters,
then — if they said 'yes' —
the handlers.



This is a call graph made of objects that each act like a verb.



This is a call graph made of objects that each act like a verb.

```
# logging.py

...

class Filter:
  def __init__(self, pattern):
    self.pattern = pattern

  def call(self, message):
    b = self.pattern in message
    return b
```

‘Stop Writing Classes’

Jack Diederich
PyCon 2012
945K views on YouTube

```
# logging.py

...

class Filter:
  def __init__(self, pattern):
    self.pattern = pattern

  def call(self, message):
    b = self.pattern in message
    return b
```

Jack:

```
# logging.py

...

class Filter:
  def __init__(self, pattern):
    self.pattern = pattern

  def call(self, message):
    b = self.pattern in message
    return b
```

Jack:

‘This is not a class.

```
# logging.py

...

class Filter:
  def __init__(self, pattern):
    self.pattern = pattern

  def call(self, message):
    b = self.pattern in message
    return b
```

Jack:

‘This is not a class.

‘It has two methods,

```python
# logging.py

...

class Filter:
  def __init__(self, pattern):
    self.pattern = pattern

  def call(self, message):
    b = self.pattern in message
    return b
```

Jack:

‘This is not a class.

‘It has two methods,
one of which is ‘__init__()’,

```
# logging.py

...

class Filter:
  def __init__(self, pattern):
    self.pattern = pattern

  def call(self, message):
    b = self.pattern in message
    return b
```

Jack:

‘This is not a class.

‘It has two methods,
one of which is ‘__init__()’,

‘and the other method
is named call().’

```
# logging.py

...

class Filter:
    def __init__(self, pattern):
        self.pattern = pattern

    def call(self, message):
        b = self.pattern in message
        return b
```

‘This is not a class.’

Rewrite as:
partial
closure

```
# logging.py

...

class Filter:
  def __init__(self, pattern):
    self.pattern = pattern

  def call(self, message):
    b = self.pattern in message
    return b
```

‘This is not a class.’

↓

This is not a noun,
this is a verb.

```python
# logging.py

...

class Filter:
  def __init__(self, pattern):
    self.pattern = pattern

  def call(self, message):
    b = self.pattern in message
    return b
```

Here’s something Jack didn’t mention when he said ‘Stop Writing Classes’:

Class instances can do things that closures and partials can’t.

```
# logging.py

...

class Filter:
  def __init__(self, pattern):
    self.pattern = pattern

  def call(self, message):
    b = self.pattern in message
    return b
```

Here’s something Jack
didn’t mention when he said
‘Stop Writing Classes’:

Class instances can do things
that closures and partials can’t.

Introspection

```
print(filter.pattern)
print(handler.filename)
```

‘logging-tree’

in the
Python Package Index

```
>>> logging_tree.printout()

<--""
   Level WARNING
   Filter name='database'
   |
   o<--"database"
       Level INFO
       Handler Stream <stdout>
```

So if ‘Encapsulation’
doesn’t just mean
‘writing a class’ —

So if ‘Encapsulation’
doesn’t just mean
‘writing a class’ —

then, when do you start
your journey into
Encapsulation?

‘Encapsulation’ starts
when what might have been a
simple data structure —

```
@dataclass
class Measurement:
  t: datetime
  temperature: float
  humidity: float
```

‘Encapsulation’ starts
when what might have been a
simple data structure —

gains a
behavior.

```
@dataclass
class Measurement:
  t: datetime
  temperature: float
  humidity: float

  def set_fahrenheit(...):
    ...
```

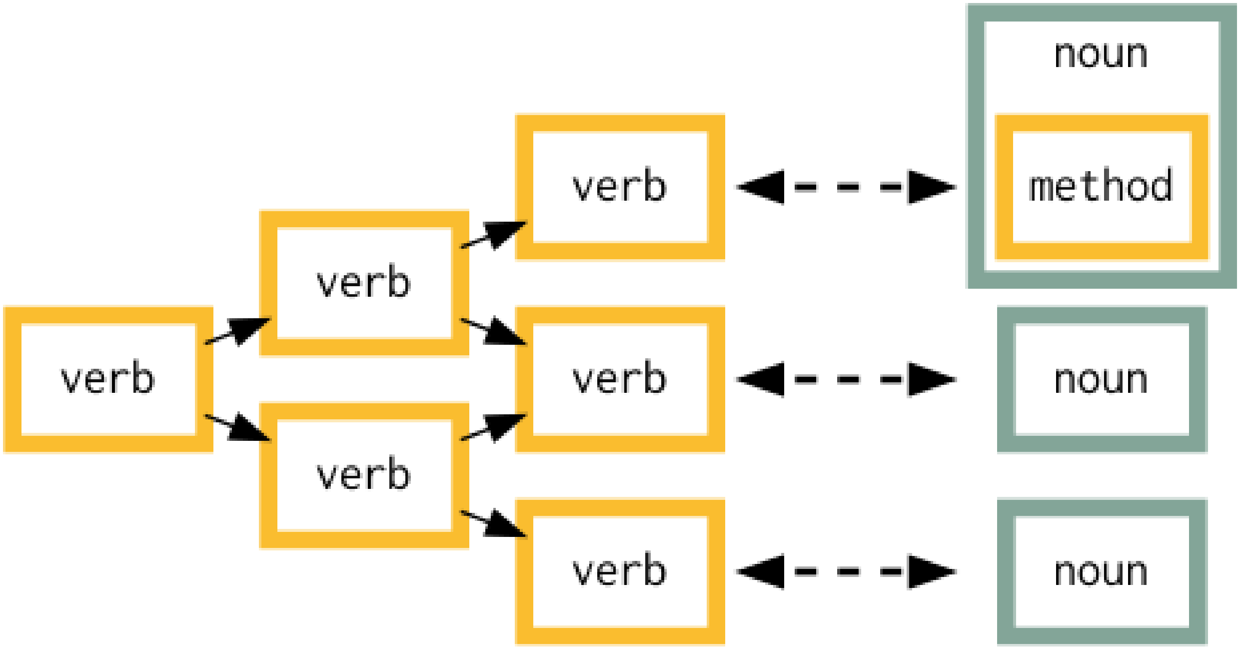
‘Encapsulation’ starts
when what might have been a
simple data structure —

gains a
behavior.

```
from django.db import models

class User(models.Model):
  name = models.CharField()
  email = models.EmailField()

  def change_email(self, addr):
    ...
```

‘Encapsulation’ starts
when what might have been a
simple data structure —

gains a
behavior.



When you embark on Encapsulation,
you become responsible for
the design of a noun.

What is a ‘Pythonic’ noun like?

Verbs

stateless
idempotent
functions

Verbs	Nouns
stateless	*stateful*
idempotent	*mutable*
functions	*data*

Verbs	Nouns
stateless	*stateful*
idempotent	*mutable*
functions	*data*
	↓
	legible
	shallow

‘legible’

No hidden state.

Nouns

stateful
mutable
data
↓
legible
shallow

‘legible’

You should be able
to print() or inspect a noun,

and predict the outcome of
every method and operation
that it supports.

Nouns

stateful
mutable
data
↓
legible
shallow

It’s fine for a
noun to hide its
implementation

but it must
never
hide its
state.

Nouns

stateful
mutable
data
↓
legible
shallow

‘shallow’

Each method on a noun should perform a single operation and then return control.

Nouns

stateful
mutable
data
↓
legible
shallow

Why should nouns be

legible
and
shallow?

Nouns

stateful
mutable
data
↓
legible
shallow

Here are two reasons.

Nouns

stateful
mutable
data
↓
legible
shallow

verbs ↔ nouns

When debugging, nouns are supposed to be the *easy part.*

Nouns

stateful
mutable
data
↓
legible
shallow

“Show me your code,
and I’ll probably be mystified.

Show me your data structures,
and I won't usually need your code;
it’ll be obvious.”

Fred Brooks (1975)
The Mythical Man-Month
(paraphrased)

Nouns

stateful
mutable
data
↓
legible
shallow

Other reason:

When staring at a .py file,
where is the code?

Right in front of you.

Nouns

stateful
mutable
data
↓
legible
shallow

Other reason:

When staring at a .py file,
where is the code?

Right in front of you.

Where is the data?

Nouns

stateful
mutable
data
↓
legible
shallow

Other reason:

When staring at a .py file,
where is the code?

Right in front of you.

Where is the data?

In your imagination.

Nouns

stateful
mutable
data
↓
legible
shallow

Other reason:

When staring at a .py file,
where is the code?

Right in front of you.

Where is the data?

In your imagination.

```
minute = timedelta(minutes=1)

def segments(t):
  tprev = t[0]
  start = 0
  for i, ti in enumerate(t):
    if ti - tprev > minute:
      yield start, i
      start = i
    tprev = ti
  yield start, i
```

Nouns need to be simple because they are *invisible half*

of reading code and reasoning about code.

```
minute = timedelta(minutes=1)

def segments(t):
  tprev = t[0]
  start = 0
  for i, ti in enumerate(t):
    if ti - tprev > minute:
      yield start, i
      start = i
    tprev = ti
  yield start, i
```

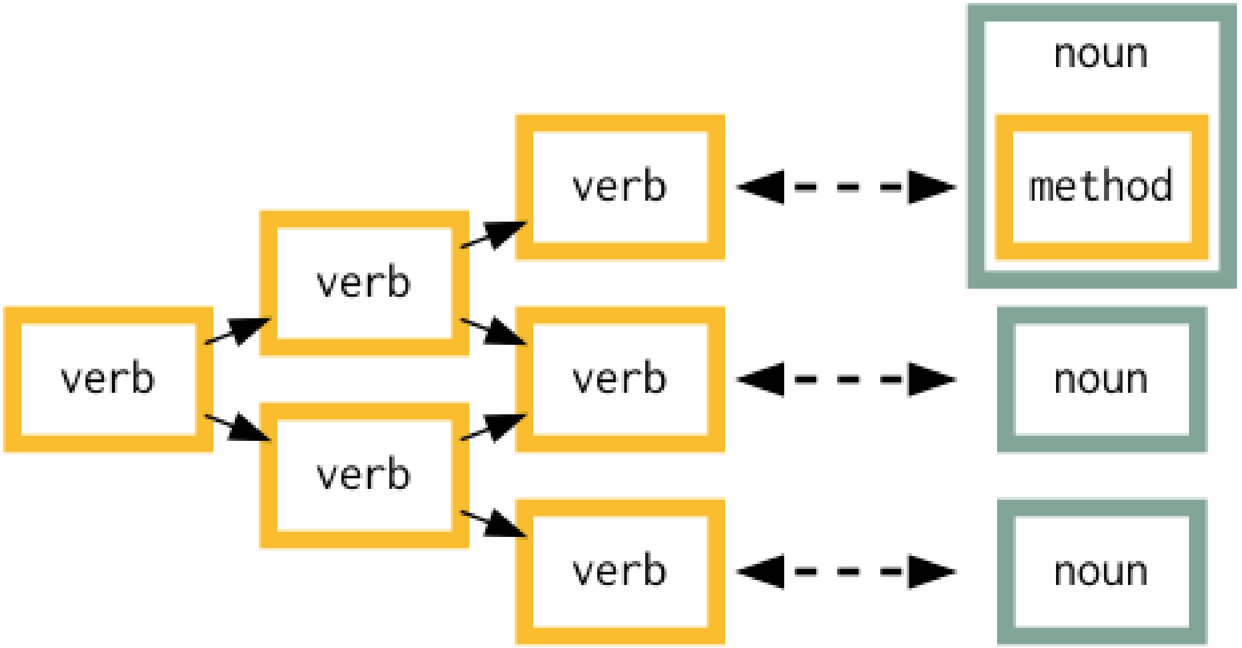
Nouns need to be simple
because they are
invisible half

of reading code and
reasoning about code.

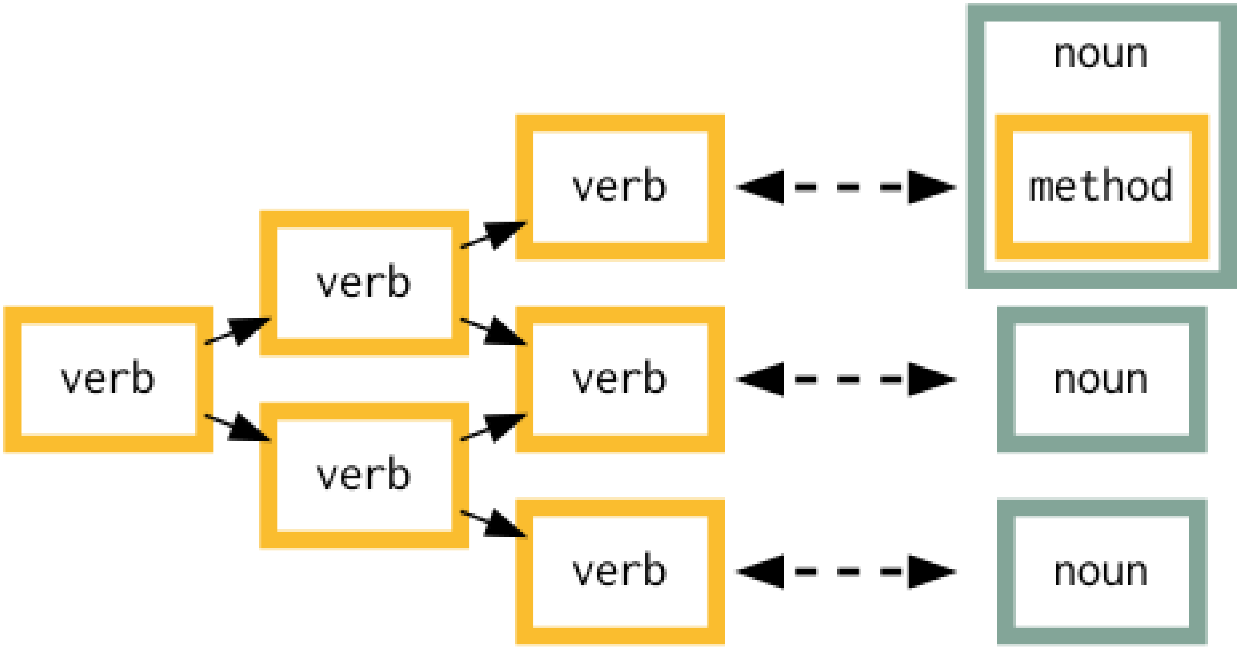
Nouns

stateful
mutable
data
↓
legible
shallow

I can now tell you what Object Orientation was.

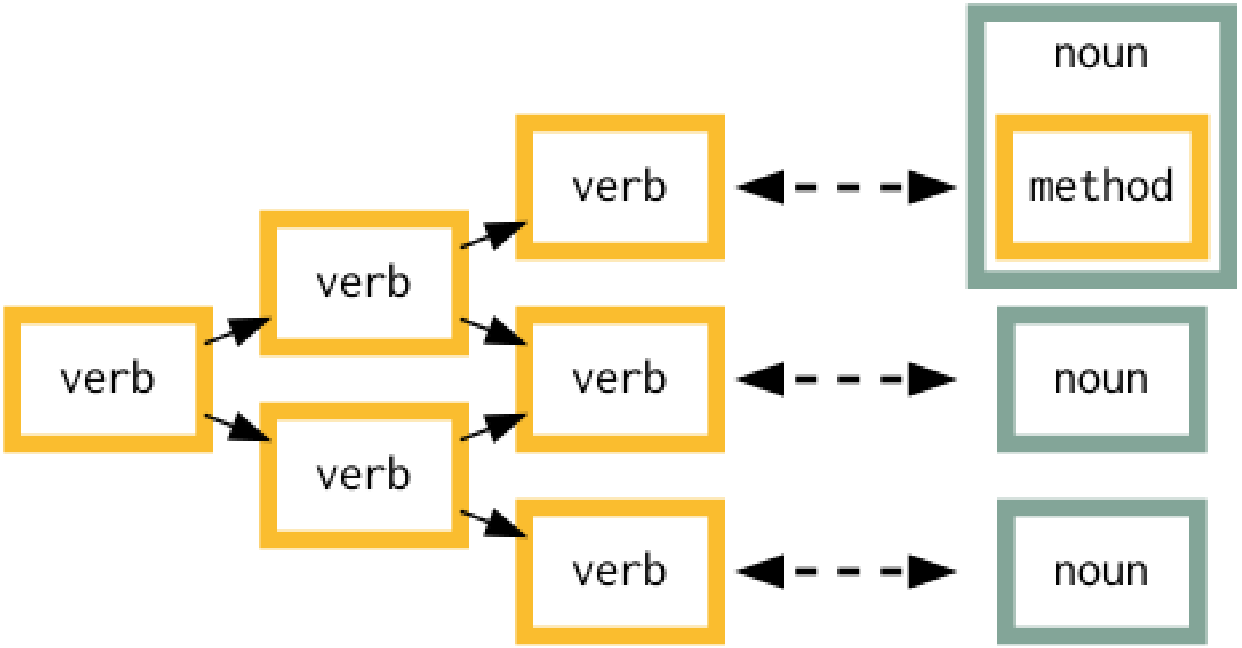


In its most academic
and pernicious form —



Object Orientation
was a ban on

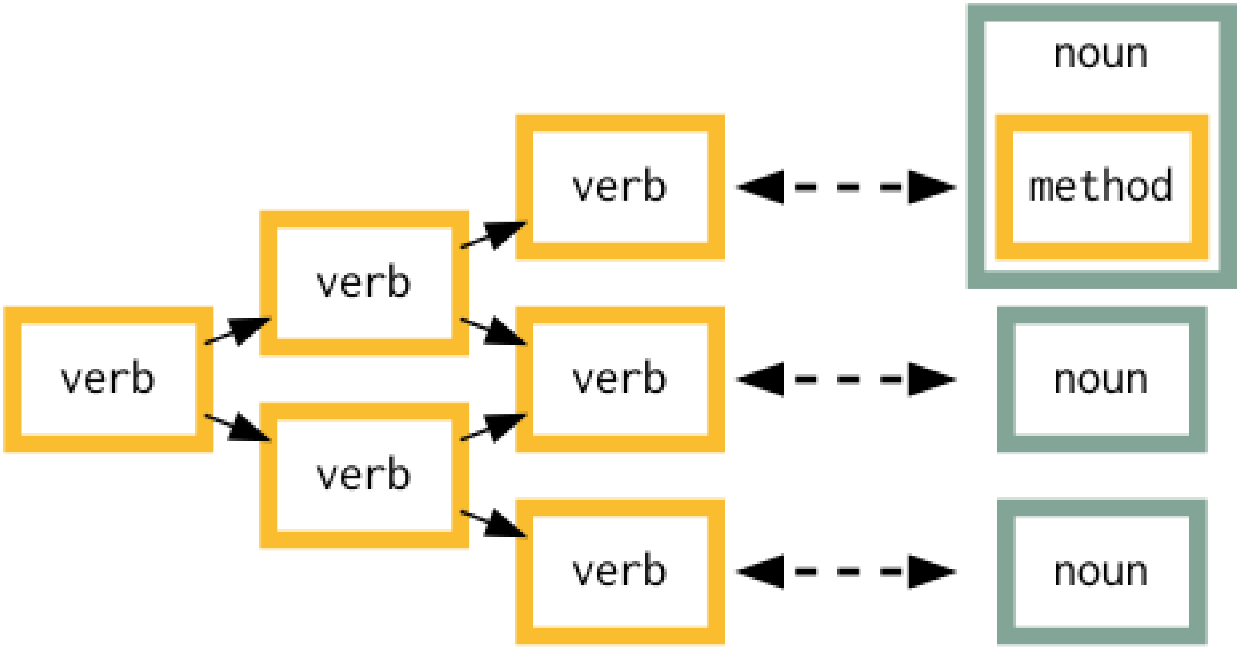
standalone verbs.



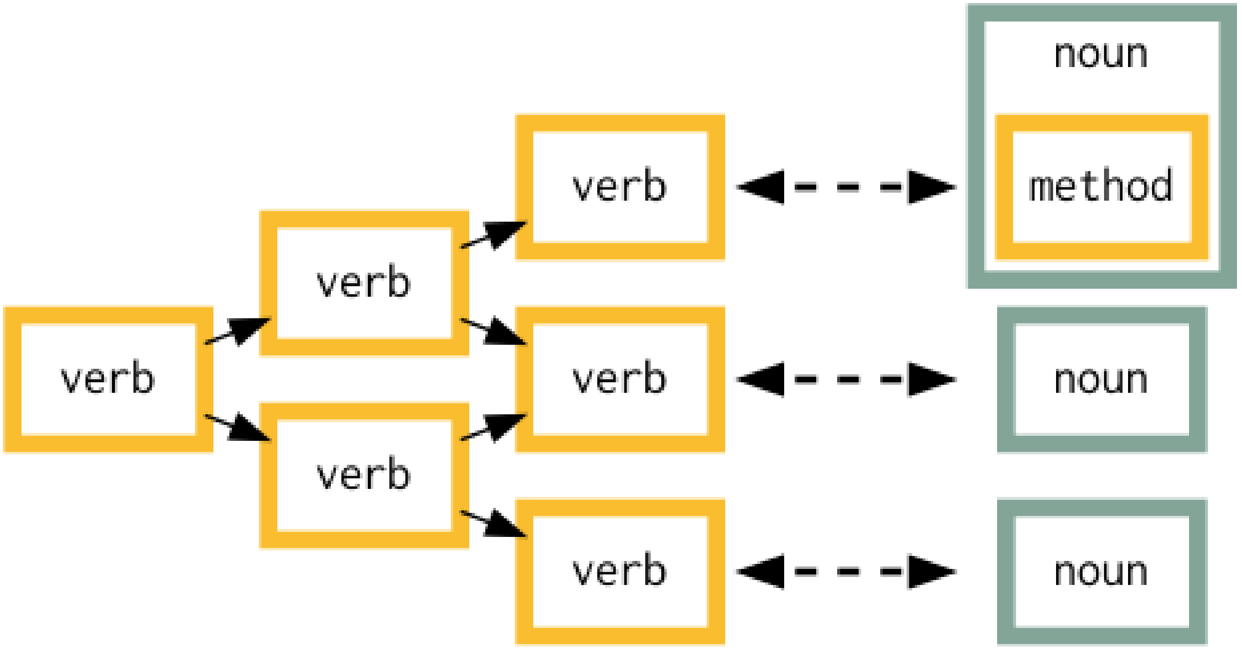
Under Object Orientation,
the left side of this diagram —

which is where I believe
most code belongs —

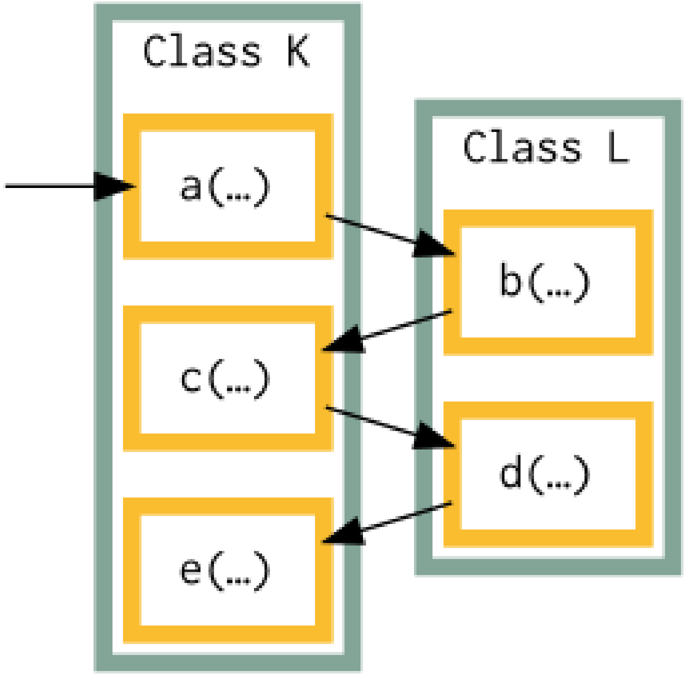
disappears.



Object Orientation recommended that you attach every line of code to the *nearest noun.*

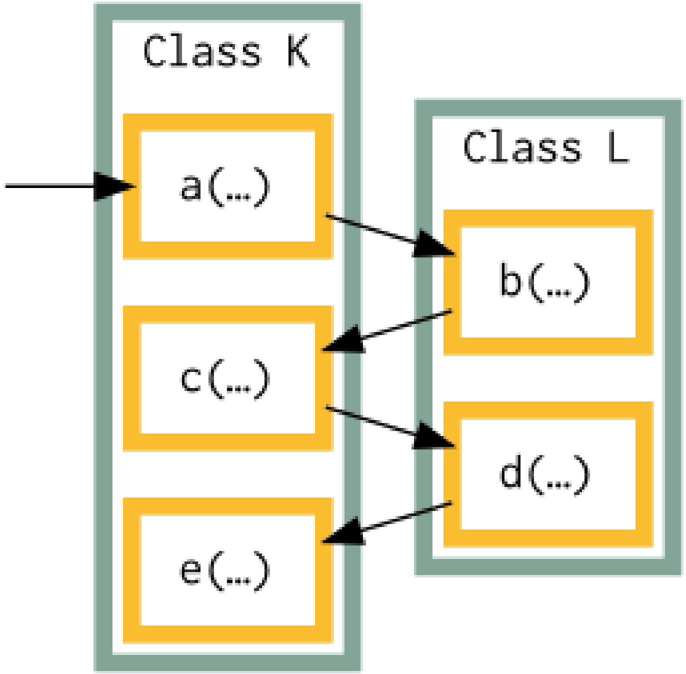


Object Orientation recommended that you attach every line of code to the *nearest noun.*

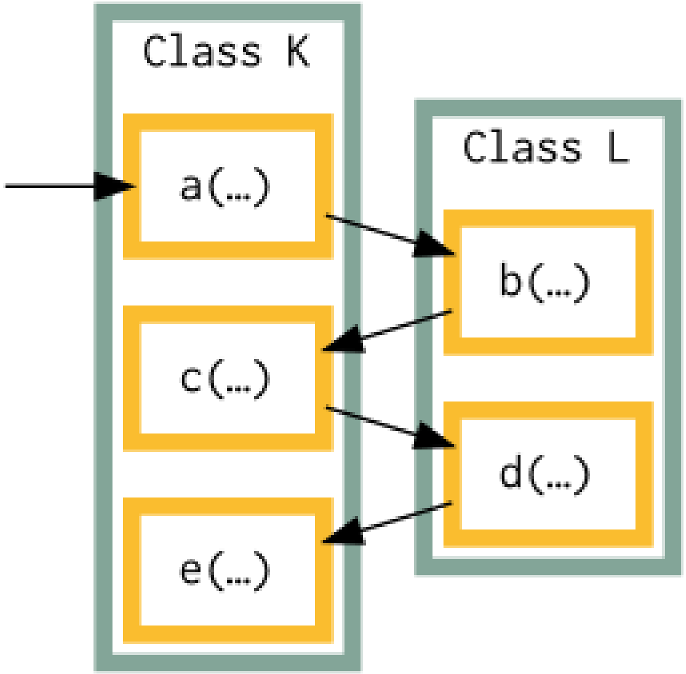


The result?

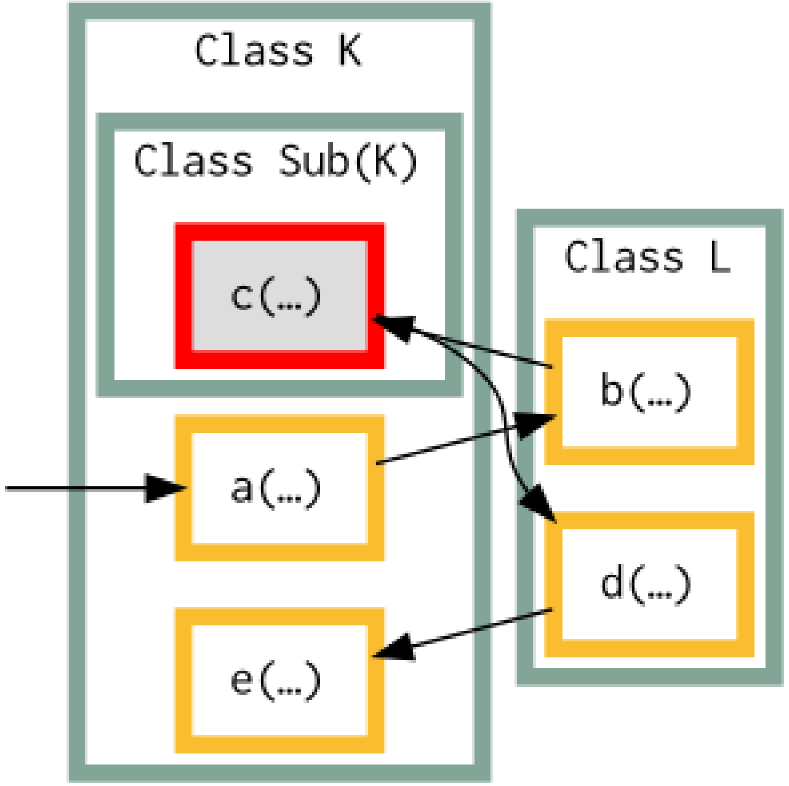
Method ping-pong.



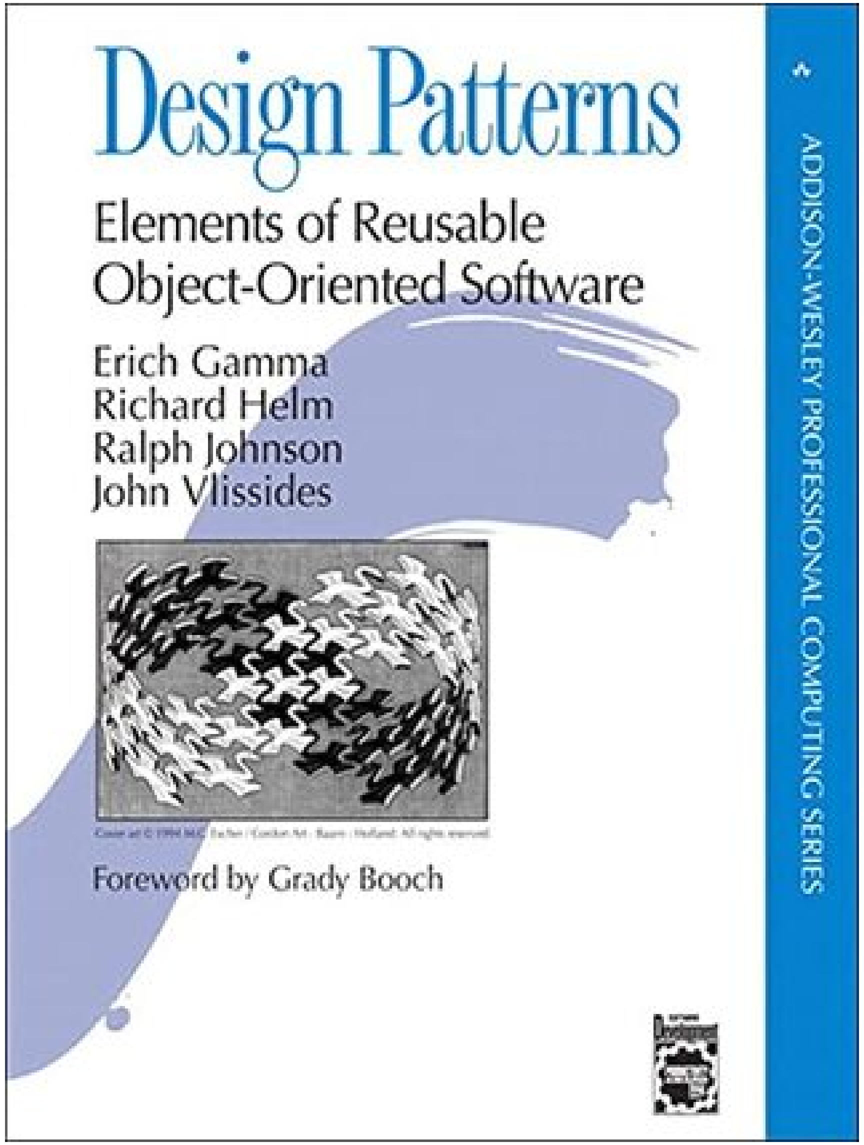
And what happens when one step needs to be adjusted?



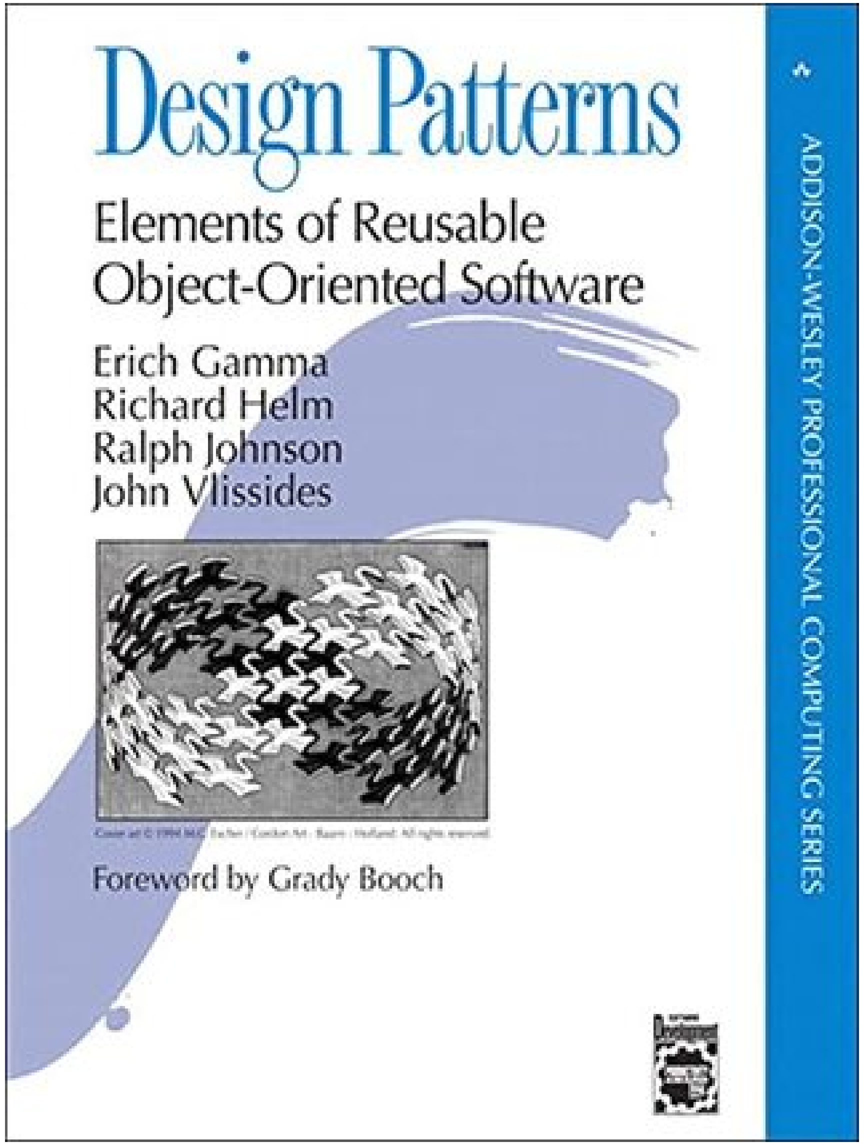
And what happens when one step needs to be adjusted?



To their credit,
the Gang of Four
knew that inheritance
was usually an anti-pattern.



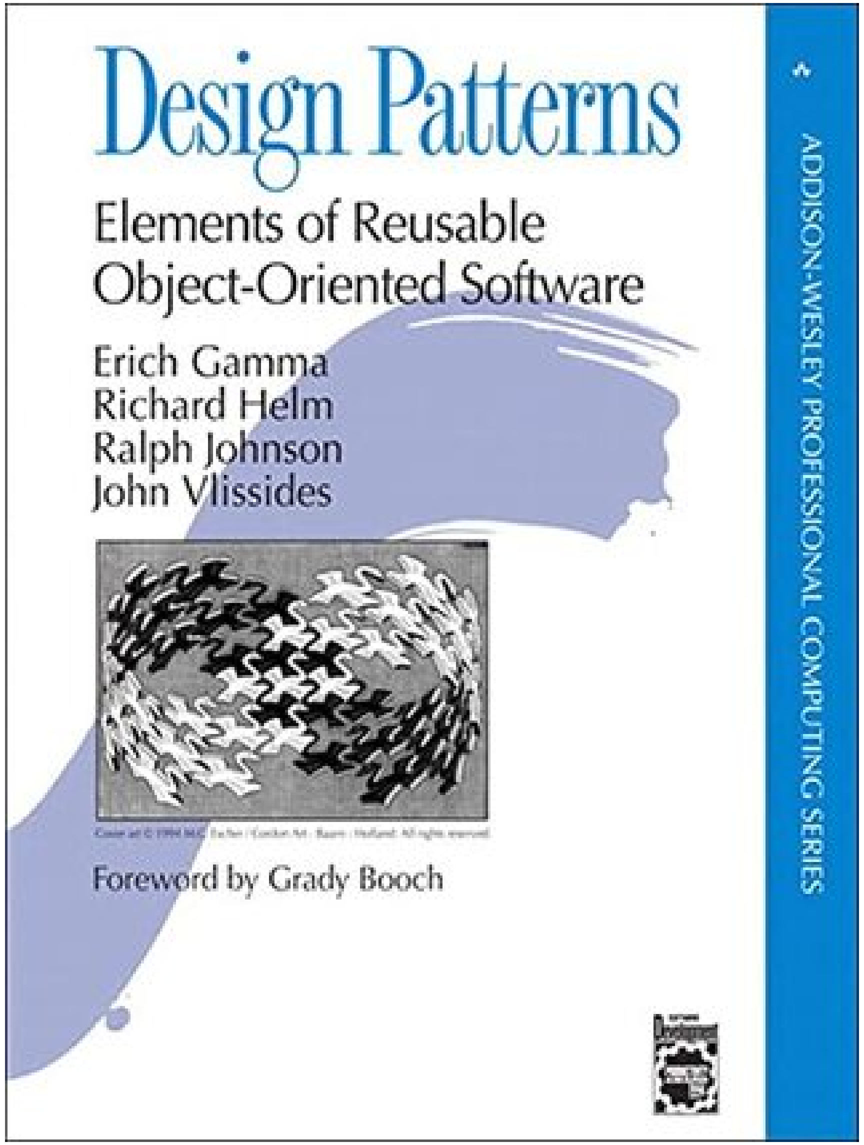
‘Favor composition
over inheritance’



Many Gang of Four
'Design Patterns'
were designed to

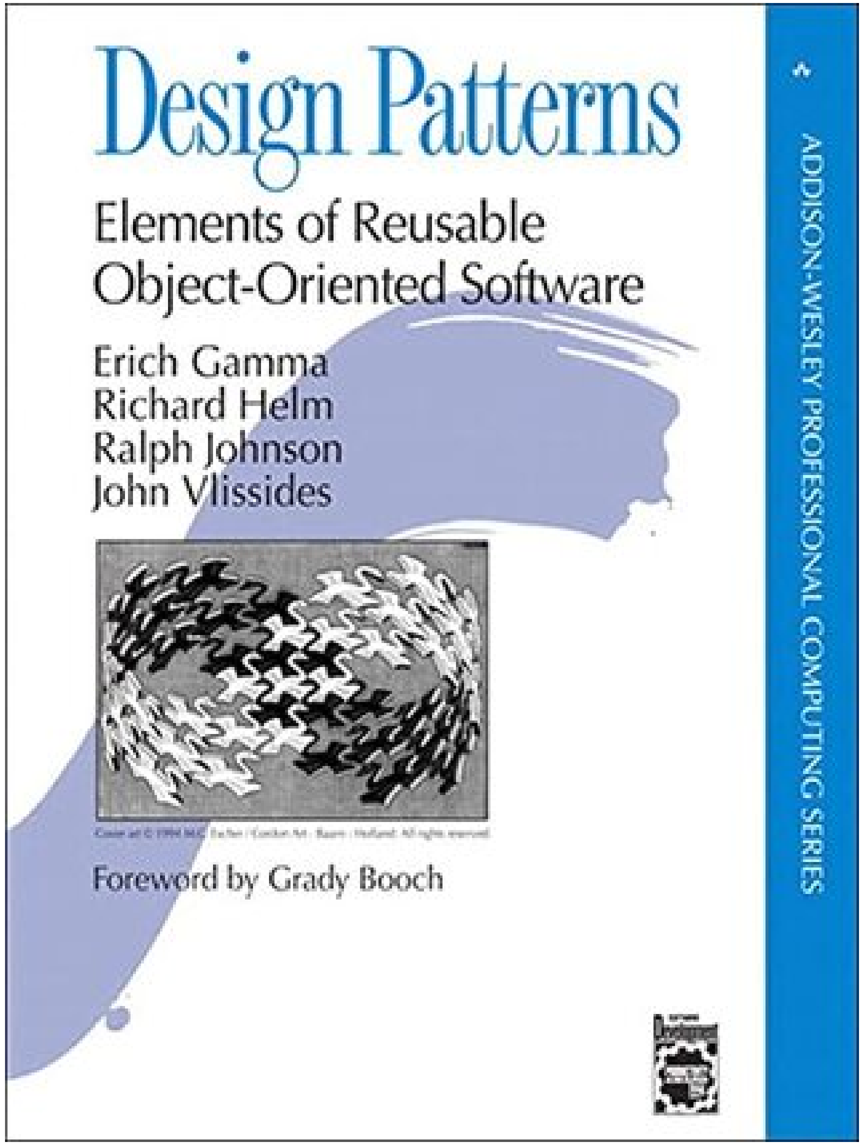
escape

from Object Orientation.



But that's why so
many Design Patterns
went away.

They could disappear
along with the bad practices
they were designed to correct.



Once you get enough
practice with ‘Encapsulation’,
you might attempt its greatest feat:

Once you get enough
practice with ‘Encapsulation’,
you might attempt its greatest feat:

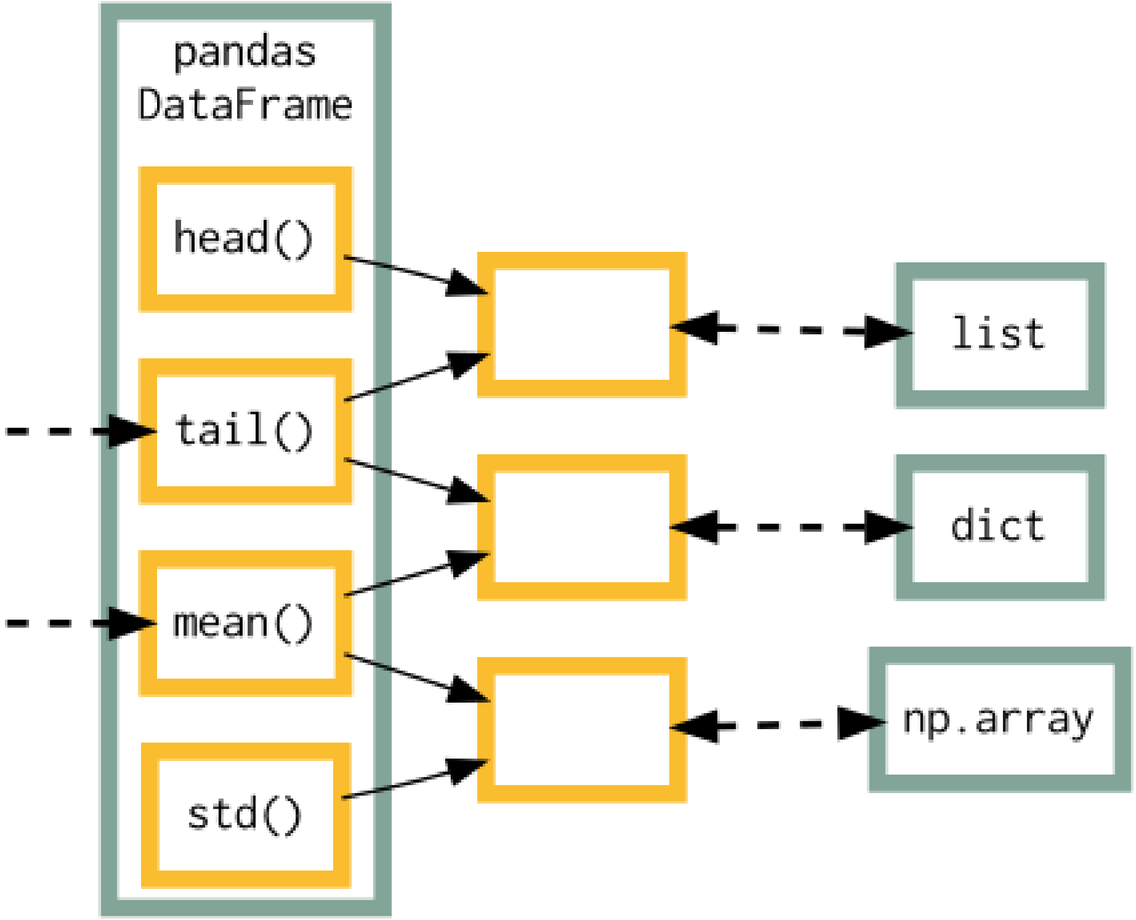
The API!

An ‘API’ hides a whole system of verbs and nouns behind an object that wraps them up as *a new noun.*

Example:
`pandas.DataFrame`

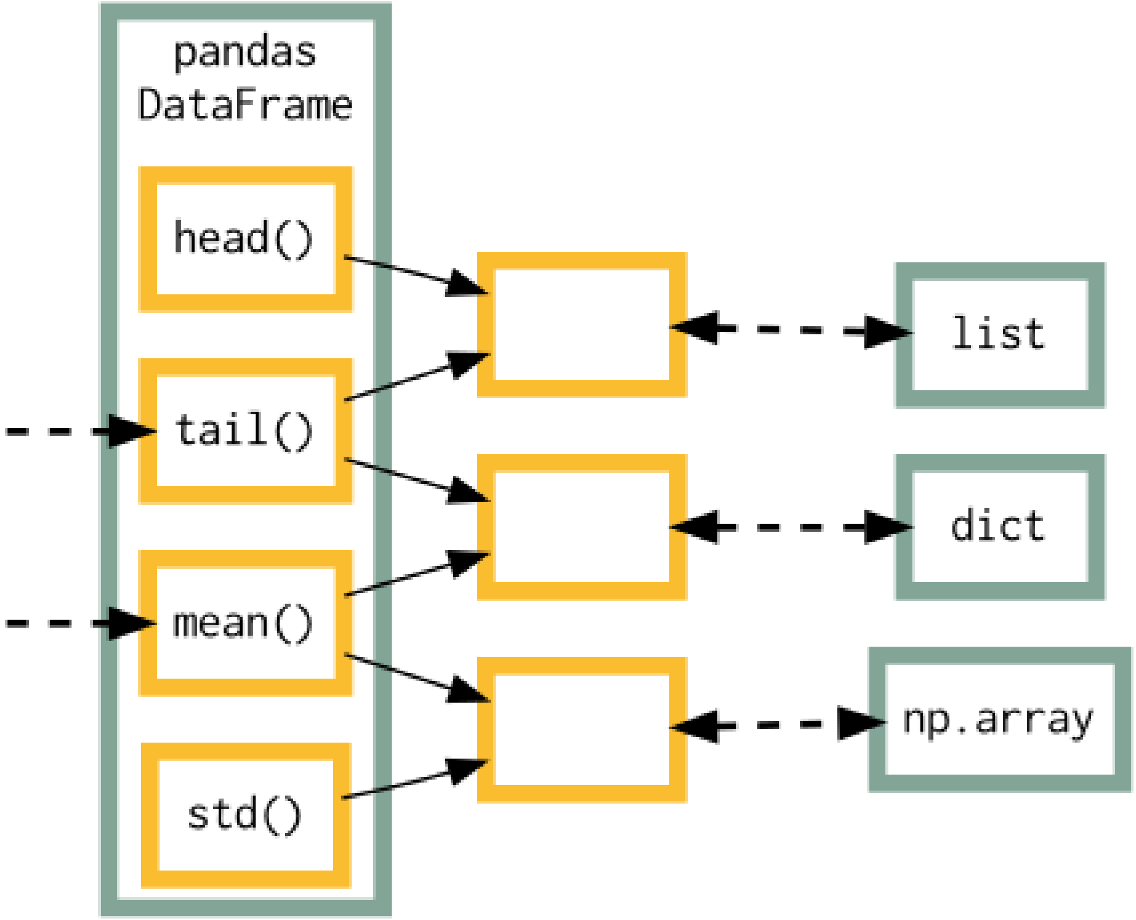
An ‘API’ hides a whole system of verbs and nouns behind an object that wraps them up as *a new noun.*

Example:
`pandas.DataFrame`

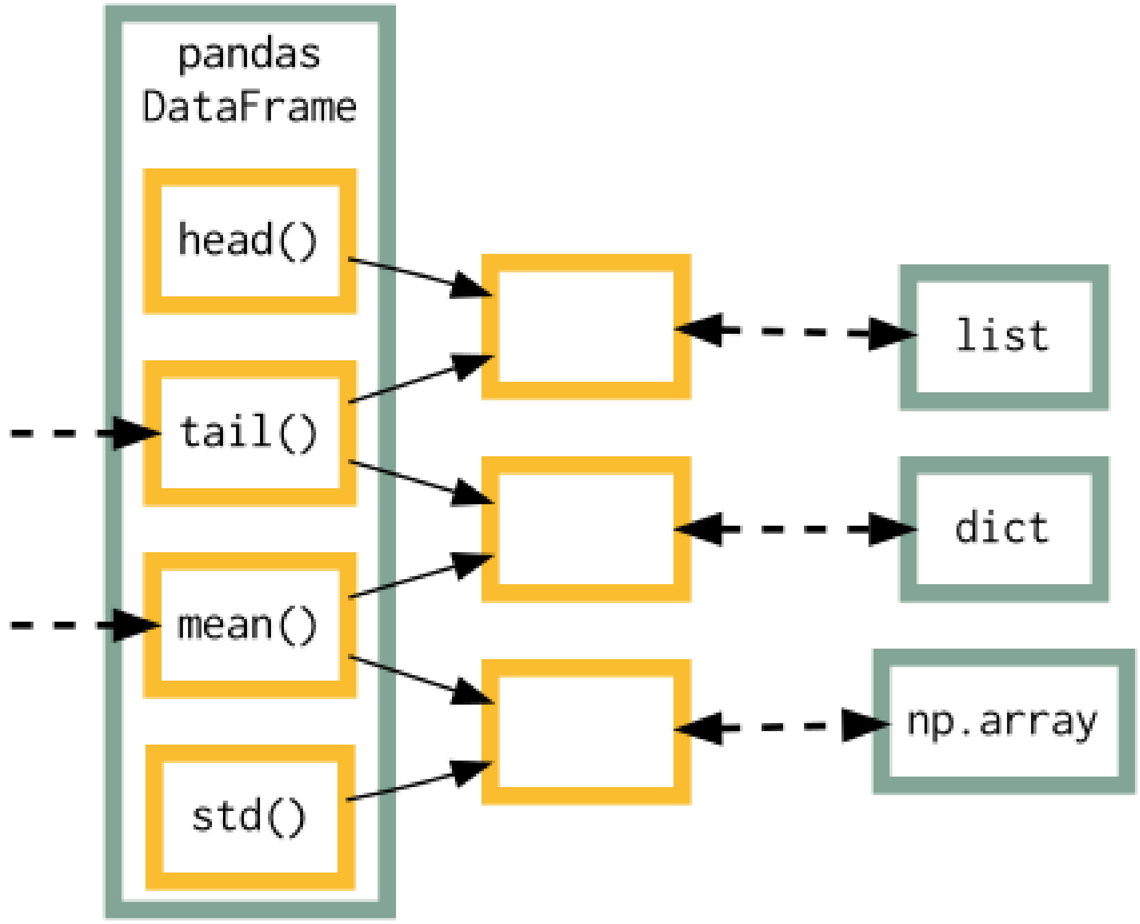


`pandas.DataFrame`
is a noun:

stateful
mutable
data
↓
legible
shallow



A real-world API
is rarely a single object —
usually it's several
that cooperate.



A real-world API
is rarely a single object —
usually it's several
that cooperate.

'Pandas'

`DataFrame` `Index` `Series`

'requests'

`Session` `Response`

But the several objects
still cooperate to maintain
common state that is

legible
and
shallow.

‘Pandas’

`DataFrame` `Index` `Series`

‘requests’

`Session` `Response`

To review —

1. Procedural ← Use nouns and verbs.
2. Abstraction ← Invent new verbs.
3. Encapsulation ← Invent new nouns.

1. Procedural ← Novice
2. Abstraction
3. Encapsulation ← Expert

1. Procedural
2. Abstraction
3. Encapsulation

Claim:

By supporting this range of approaches to writing code, Python helps you *learn.*

Your first ‘Procedural’
code might all be in main() —

But you’re getting practice
using nouns and verbs
written by experts.

That experience with using ‘Abstractions’
might help you design functions of your own.

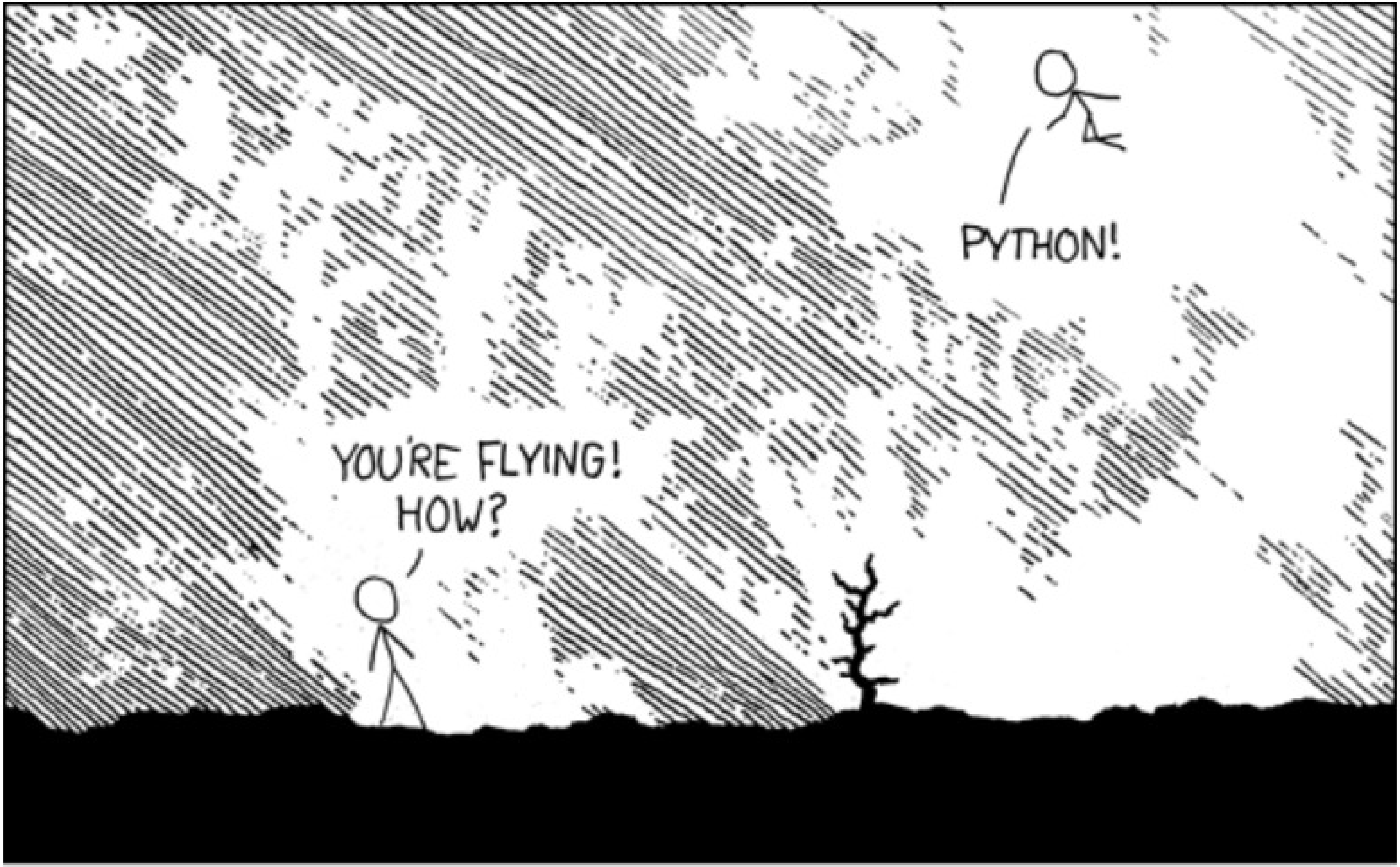
If someday you write your own API,
you'll use your experience with how
other libraries do 'Encapsulation'.

In conclusion:
by letting you use nouns and verbs
before you build them yourself —

Python lets you
walk

before you
run

before you
fly.


YOU'RE FLYING!
HOW?
PYTHON!