py2store

A DAO of Python

Tools to create simple & consistent interfaces to/from complicated & varied data systems.

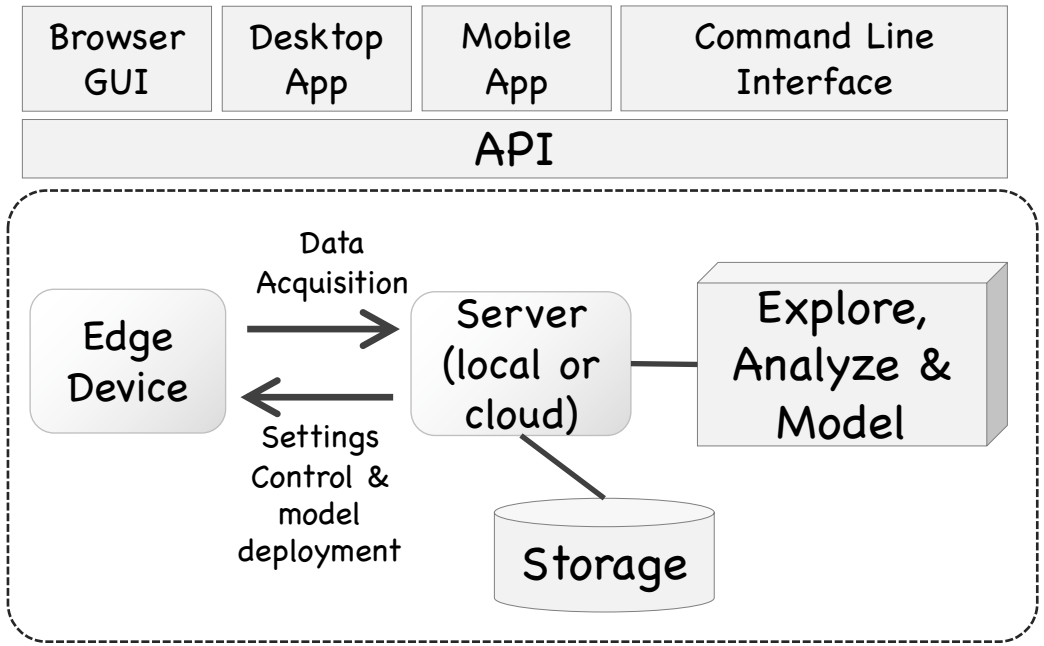
Thor Whalen. Director of AI Research at Analog Devices
email: thor.whalen at analog
py2store repo: https://github.com/i2mint/py2store
A demo notebook for this talk: https://github.com/i2mint/examples/tree/master/pybay_2019

You can't snore and dream at the same time. (*)

(*) http://davitzinc.blogspot.com/2015/07/fun-facts-can-you-snore-and-dream-at.html

Borelerplate and other creativity killers

They (*) say **80%** of a data scientists **time on data preparation**.
Data prep? Accessing, cleaning, restructuring, generating more directly usable data.



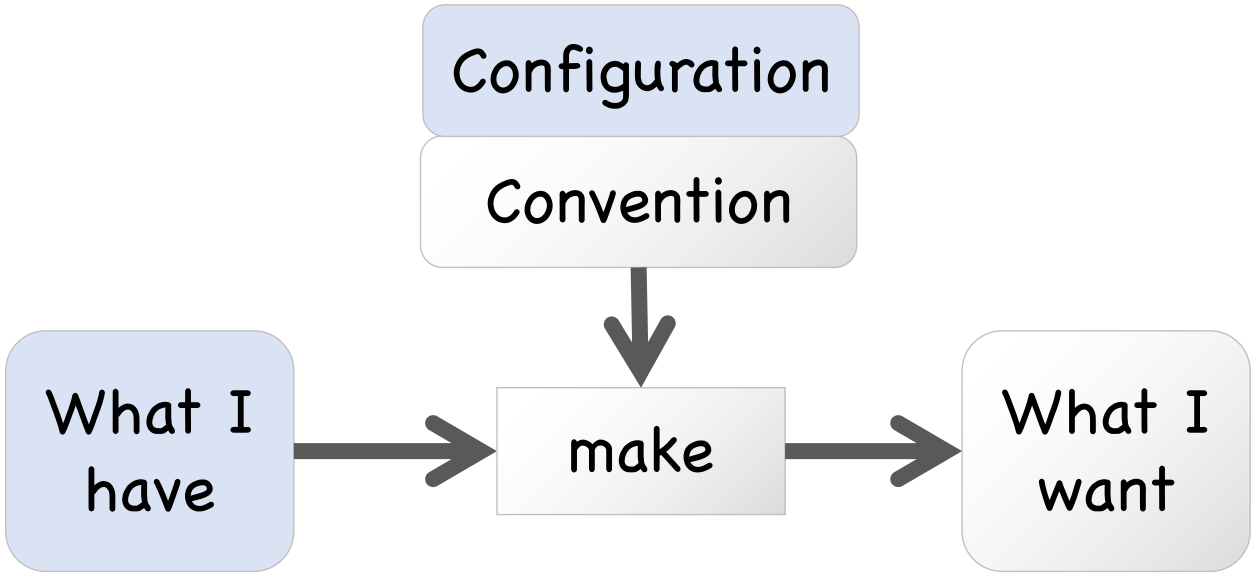
Once the data scientist did the extra 20% and is done, her work **needs to be integrated**...

Integrated in backend processes.
Provide a CLI
Make a web-service for it.
Provide a GUI

... and adapt to constant, never ending, incorrigible **CHANGE**.

(*) the internet

Does this process look familiar?



py2ui

```python
def foo(a: int = 0, b: int = 0, c=0):
    return (a * b) + c


def bar(x, greeting='hello'):
    return f"{greeting} {x}"


def confuser(a: int = 0, x: float = 3.14):
    return (a ** 2) * x
```

py2ui

```
funcs = [foo, bar, confuser]

if __name__ == '__main__':
    from py2dash.app_makers import dispatch_funcs
    from py2dash.conventions import raw_style  # import a convention
    raw_style.func_navigs = 'links'  # modify or configure the convention
    app = dispatch_funcs(funcs, raw_style)  # make a website
    app.run_server(debug=True)  # launch it
```

py2ui

foo

a

3

b

4

c

-5

7

Home

Bar

Confuser

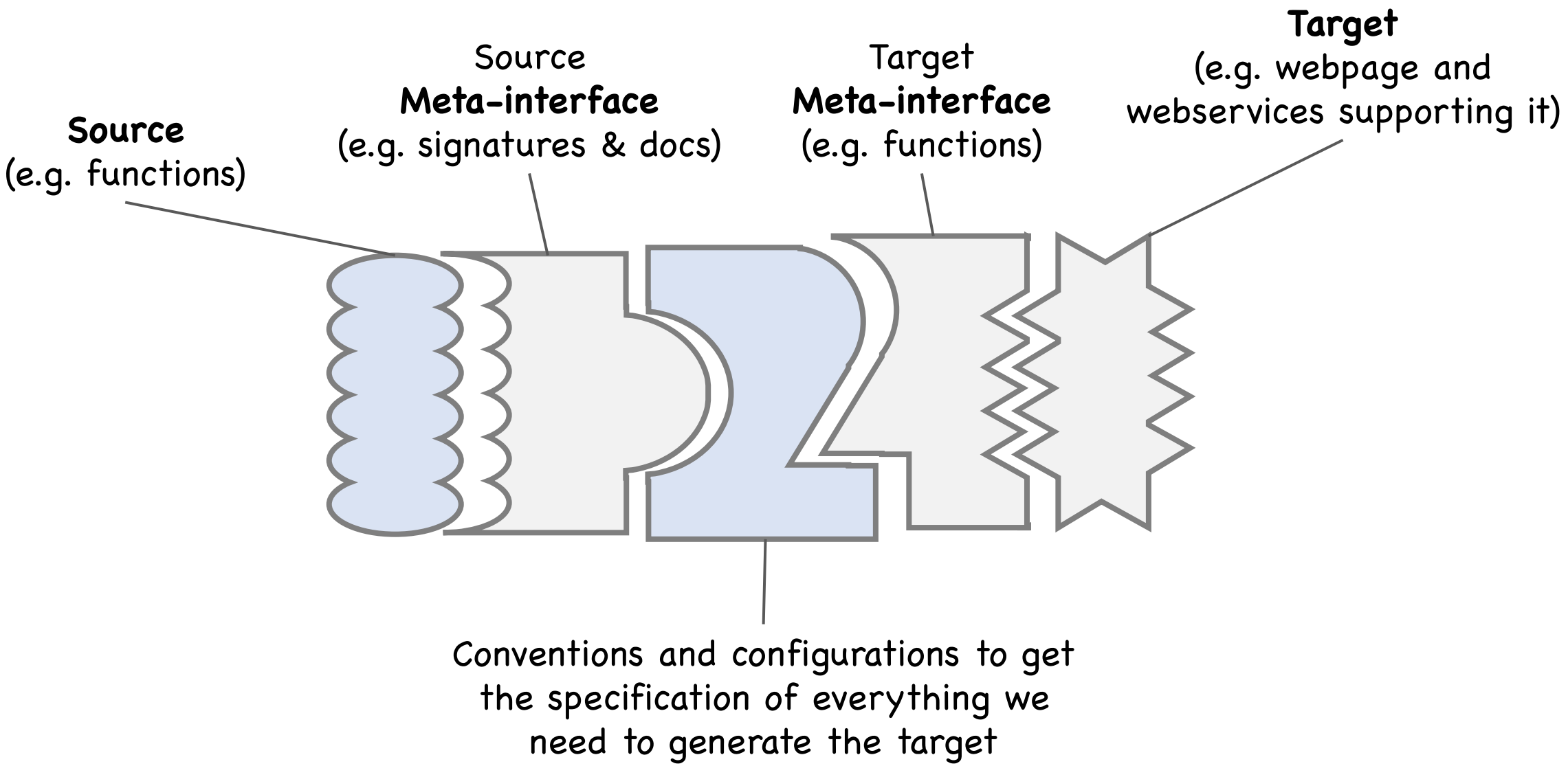
py2ws (when it's easy)

```
from mystuff import SystemService
from py2api import OutputTrans
from py2api.py2rest import InputTransWithAttrInURL

attr_list = ['ping', 'status', 'stop', 'start', 'restart']

name = '/api/process/'
process = WebObjWrapper(
    obj_constructor=SystemService,
    obj_constructor_arg_names=['name'],
    permissible_attr=attr_list,
    input_trans=InputTransWithAttrInURL(trans_spec=None, attr_from_url=name + "(\w+)"),
    output_trans=OutputTrans(trans_spec=None),
    name=name,
    debug=0
)

app = add_routes_to_app(app, routes={
        process.__name__ + attr: process for attr in attr_list
    }
)
```

py2ws (when it's harder)

```
from py2api.constants import _ATTR, _ARGNAME, _ELSE, _VALTYPE

def ensure_array(x):
    if not isinstance(x, ndarray):
        return array(x)
    return x

trans_spec = {
        _ATTR: {
            'this_attr': list,
            'other_attr': str,
            'yet_another_attr': {
                _VALTYPE: {
                    dict: lambda x: x
                },
                _ELSE: lambda x: {'result': x}
            }
        },
        _ARGNAME: {
            'sr': int,
            'ratio': float,
            'snips': list,
            'wf': ensure_array
        },
    }
}
```

The interface-to-interface view



7 ± 2 (*)

(*) Miller, G. A. (1956). "The magical number seven, plus or minus two: Some limits on our capacity for processing information". Psychological Review (CiteSeerX 10.1.1.308.8071)

New words to make computers easier to use (*)

(*) Raymond Hettinger (PyBay 2019)

Based on a true story...

0.1 Import the basic stuff you'll need

```
# Sound: Holds a waveform and sample rate and displays nicely (and so that sound can be played)
# MockAudioBuffer: A pretend sensor buffer (sourcing in wav files)
from pybay2019 import *
```

executed in 21ms, finished 11:04:01 2019-08-17

```
print(f"{Sound.__name__}: {Sound.__doc__})\n")
print(f"{MockAudioBuffer.__name__}: {MockAudioBuffer.__doc__})\n")
print(f"{do_something_with_stream.__name__}: {do_something_with_stream.__doc__})\n")
```

executed in 22ms, finished 11:04:03 2019-08-17

```
Sound: Holds a waveform and sample rate and displays nicely (and so that sound can be played))

MockAudioBuffer: A pretend "sensor buffer". It sources from wav files and returns fixed sized 'time-stamped chunks' o
n read.
    Time-stamped chunks are given as (session_utc_time, block_byte_offset, data))

do_something_with_stream: Reads through a t = (session, block, data) timestamped chunks stream and calls the function
'something' on t.)
```

Based on a true story...

0.2 A function to give us random signal streams

```
wavs_dir = 'data/wavs_16bit'
wjoin = lambda p: os.path.join(wavs_dir, p)
wav_files = list(filter(lambda x: x.endswith('.wav'), os.listdir(wjoin(''))))

def get_random_stream(chk_size_bytes=DFLT_CHK_SIZE_BYTES):
    wav_file = wjoin(np.random.choice(wav_files))
    return MockAudioBuffer(wav_file)
```

executed in 24ms, finished 11:02:30 2019-08-17

The raw signal is actually fed through a buffer that releases `(session, block, chk)` triples, where

- chk is a fixed size chunk of bytes (from the sensor)
- block is the byte index of that chunk (equivalently, the number of bytes that have been popped before this chunk, during this "session"
- session is the (UTC seconds) timestamp of when a capture session started,

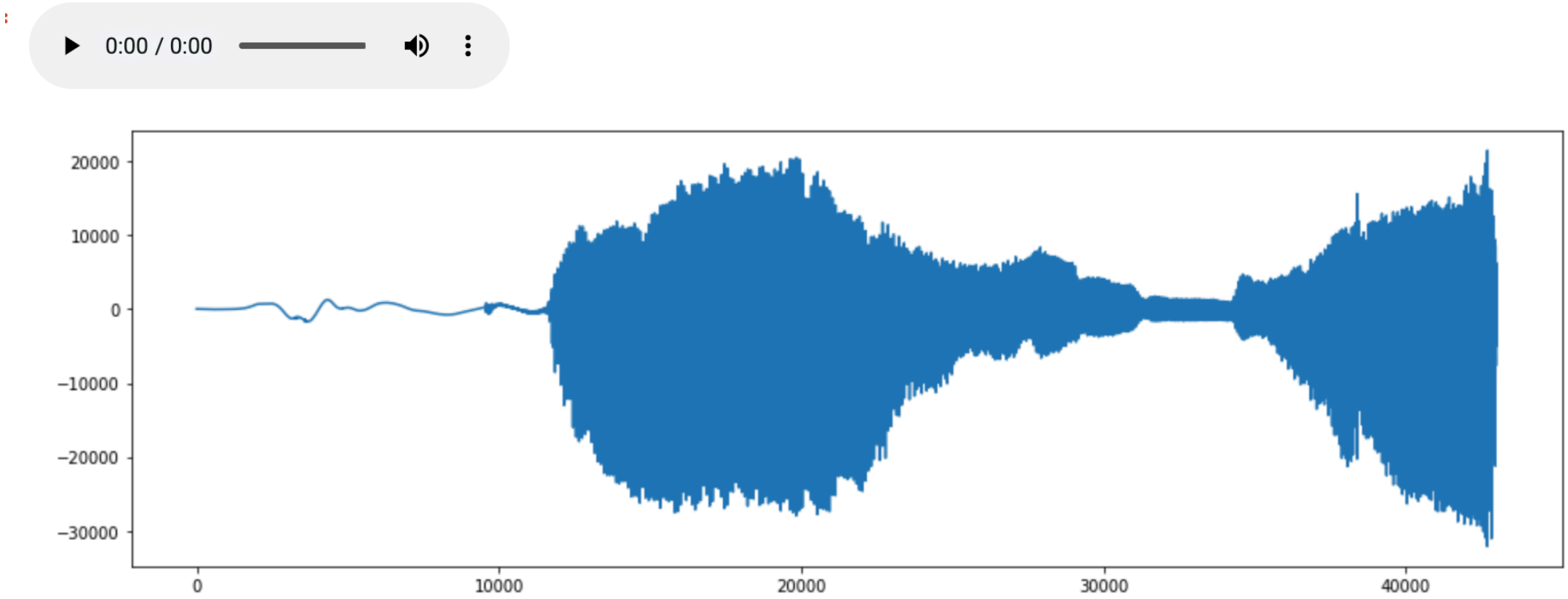
Let's have a look at this data...

Based on a true story...

```
with get_random_stream() as source:
    sr = source.sr  # remember the sample rate
    session, block, chk = source.read()

print(f"session={session}, block={block}")
# get a waveform from the chk (bytes) and display it (and listen to it)
wf, sr = sf.read(BytesIO(chk), samplerate=sr, channels=1,
                 subtype='PCM_16', dtype='int16', format='RAW',  endian=None)
Sound(wf, sr).display()
```

executed in 320ms, finished 11:04:40 2019-08-17

```
session=1566065080.1502929, block=0
```



Based on a true story...

0.3 Do something with this (session, block, chk) stream...

0.3.1 Something useless: Printing (session, block)

```
:
def print_session_and_offset(session, block, chk):
    print(int(session * 1000), block)

do_something_with_stream(get_random_stream(), print_session_and_offset)
```

executed in 25ms, finished 11:04:53 2019-08-17

```
source: data/wavs_16bit/machine_gun_02.wav, sr: 44100
1566065093213 0
1566065093213 86016
1566065093213 172032
```

Based on a true story...

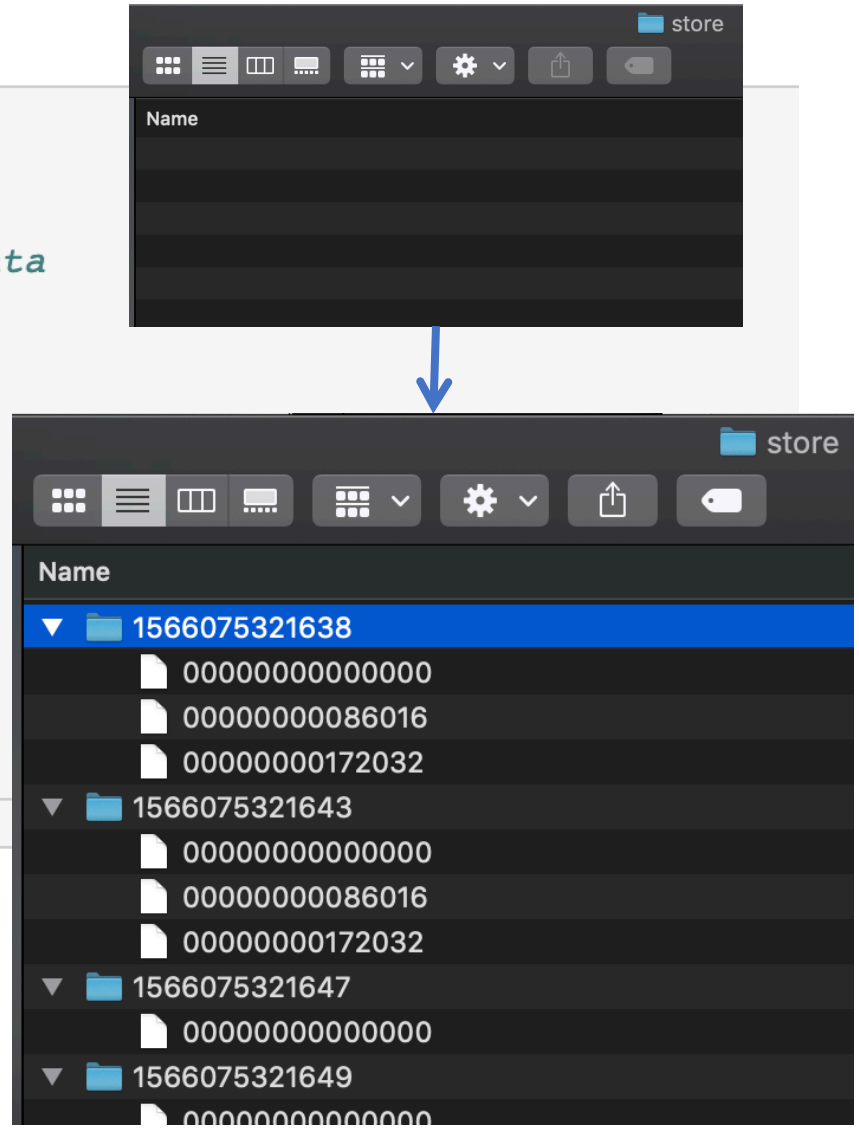
0.3.2 Something useful: Storing the signal

Let's store the signal in the file system. We choose to store the `chk` of bytes given by the `(SESSION, BLOCK, chk)` triple under `ROOTDIR/SESSION/BLOCK`:

```
:
storage_root = 'data/store/'

def store_stream(session, block, chk):
    # use session and offset to make a filepath to store the data
    filename = f"{int(session * 1000)}/{block:014.0f}"
    filepath = os.path.join(storage_root, filename)
    # make the necessary dirs!
    dirpath = os.path.dirname(os.path.abspath(filepath))
    os.makedirs(dirpath, exist_ok=True)
    # save the data
    with open(filepath, 'wb') as fp:
        fp.write(chk)

for i in range(5):
    do_something_with_stream(get_random_stream(), store_stream)
```

executed in 40ms, finished 13:55:21 2019-08-17

```
source: data/wavs_16bit/voice_05.wav, sr: 44100
source: data/wavs_16bit/machine_gun_01.wav, sr: 44100
source: data/wavs_16bit/voice_01.wav, sr: 44100
source: data/wavs_16bit/voice_05.wav, sr: 44100
source: data/wavs_16bit/voice_03.wav, sr: 44100
```



Based on a true story...

0.3.3 Retrieving the signal data

```
storage_root = 'data/store/'

def get_chk_for_session_block(session, block):
    # use session and offset to make a filepath to store the data
    if not isinstance(block, str):
        block = f"{block:014.0f}"
    filename = f"{session}/{block}"
    filepath = os.path.join(storage_root, filename)
    with open(filepath, 'rb') as fp:
        b = fp.read()
    return b
```

executed in 23ms, finished 13:54:12 2019-08-17

List the sessions

```
sessions = os.listdir(storage_root)
sessions
```

executed in 29ms, finished 10:02:04 2019-08-16

```
['.DS_Store',
 '1565917843025',
 '1565916811345',
 '1565915064799',
 '1565915044290',
 '1565915103869']
```

List the blocks within a session

```
os.listdir(os.path.join(storage_root, sessions[-1]))
```

executed in 28ms, finished 10:02:05 2019-08-16

```
['00000000000000',
 '00000000258048',
 '00000000516096',
 '00000000430080',
 '00000000172032',
 '00000000086016',
 '00000000344064']
```

Can you spot all the places where the storage concern is involved?

0.4 Do something useful with the stored signal: Retrieve waveforms...

```python
import re

storage_root = 'data/store/'


session_pattern = re.compile('\d+')
block_pattern = re.compile('\d{14}')  # tied to the {block:014.0f} format used in the filepaths
DFLT_SR = 44100
DFLT_BYTES_PER_SAMPLE = 2
time_units_per_sec = 1000

inf = float('infinity')

def coverage(a, bt=-inf, tt=inf, is_sorted=False):
    """ Subarray of (Sorted!!) array (of numbers indexing segments) that overlaps with [bt, tt)"""
    if not is_sorted:
        a = np.sort(a)
    else:
        a = np.array(a)
    greater_than_bt_idx = np.where(a >= bt)[0]
    less_than_tt_idx = np.where(a < tt)[0]
    if not any(greater_than_bt_idx) or not any(less_than_tt_idx):
        return np.array([])

    first_idx = max(0, greater_than_bt_idx[0] - 1)
    last_idx = less_than_tt_idx[-1] + 1
    return a[first_idx:last_idx]


def sessions_between(bt=-inf, tt=inf):
    sessions = list(map(int, filter(session_pattern.match, os.listdir(storage_root))))
    return coverage(sessions, bt, tt)

def session_sorted_blocks(session):
    return sorted(map(int, filter(block_pattern.match,
                                  os.listdir(os.path.join(storage_root, str(session))))))
```

```python
def get_wfsr_between(bt=-inf, tt=inf, sr=DFLT_SR, bytes_per_sample=DFLT_BYTES_PER_SAMPLE):
    """ An iterator of waveforms (one per sessions overlapping with [bt, tt) interval)
    Note: for simplicity, we're returning the "block level" coverage of [bt, tt),
    which will always be a bit more than the sample-level one.
    """
    _wfsr_from_bytes = partial(wfsr_from_bytes, sr=sr, bytes_per_sample=bytes_per_sample)
    current_session = None
    current_session_bytes = b''
    for session, block in blocks_between(bt, tt, sr, bytes_per_sample):
        if session != current_session:
            if len(current_session_bytes) > 0:
                yield _wfsr_from_bytes(current_session_bytes)
            current_session = session
            current_session_bytes = b''  # reset bytes cumul
        current_session_bytes += get_chk_for_session_block(str(session), f'{block:014.0f}')
    yield _wfsr_from_bytes(current_session_bytes)
```

```python
def blocks_between(bt=-inf, tt=inf, sr=DFLT_SR, bytes_per_sample=DFLT_BYTES_PER_SAMPLE):
    def byte_offset_from_utc_ms(utc_ms, utc_ms_of_byte_zero):
        return ((utc_ms - utc_ms_of_byte_zero) / time_units_per_sec) * sr * bytes_per_sample

    sessions = sessions_between(bt, tt)
    n_sessions = len(sessions)
    if n_sessions > 1:

        session = sessions[0]

        bt_in_byte_offset = byte_offset_from_utc_ms(bt, session)
        blocks = coverage(session_sorted_blocks(session), bt=bt_in_byte_offset, is_sorted=True)
        yield from zip([session] * len(blocks), blocks)

        for session in sessions[1:-1]:
            blocks = session_sorted_blocks(session)
            yield from zip([session] * len(blocks), blocks)

        session = sessions[-1]
        bt_in_byte_offset = byte_offset_from_utc_ms(bt, int(session))
        blocks = coverage(session_sorted_blocks(session), bt=bt_in_byte_offset, is_sorted=True)
        yield from zip([session] * len(blocks), blocks)

    elif n_sessions == 1:
        session = sessions[0]
        blocks =  coverage(session_sorted_blocks(session),
                           bt=byte_offset_from_utc_ms(bt, session),
                           tt=byte_offset_from_utc_ms(tt, session),
                           is_sorted=True)
        yield from zip([session] * len(blocks), blocks)
    else:
        return []

from functools import partial

read_kwargs_for_bps = {
    2: {'subtype': 'PCM_16', 'dtype': np.int16},
    3: {'subtype': 'PCM_24', 'dtype': np.int32},
    4: {'subtype': 'PCM_32', 'dtype': np.int32}}

read_kwargs = dict(channels=1, format='RAW', endian=None)

def wfsr_from_bytes(b, sr=DFLT_SR, bytes_per_sample=DFLT_BYTES_PER_SAMPLE):
    if bytes_per_sample in read_kwargs_for_bps:
        return sf.read(BytesIO(b), samplerate=sr,
                       **dict(read_kwargs, **read_kwargs_for_bps[bytes_per_sample]))
    else:
        raise ValueError(f'bytes_per_sample must be 2, 3, or 4 (was {bytes_per_sample})')
```

0.5 Change (and s**t) happens ... and then, you need to adapt

0.5.1 Need to add information about sessions...

And things go onwards as such until one day...

A client wants to be able to add some information about the sessions. Many ways to do this. The way that was chosen is to continue using the file system, organized as such:

```
ROOTDIR/SESSION/data/BLOCK  # where the signal chunks should now be stored
ROOTDIR/SESSION/info/       # where the client's various files about the session will be stored
```

So we just need to copy the data over to this kind of template, and change our code to use this new template.

But there's so many places we need to change this! Mistakes will be made. Bugs will be born. Hopefully we'll discover them before the client!

0.5.2 Need separate different sources

And then one day...

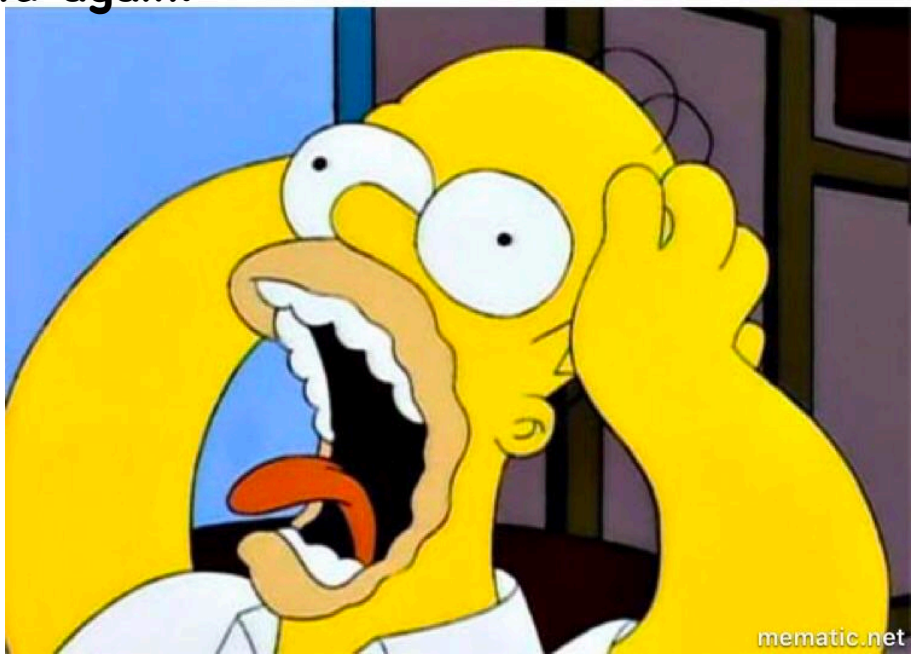
We want to enhance our senses. How can we accommodate for different sensor streams?

Well, just use this file structure now:

```
ROOTDIR/SESSION/data/SENSOR_ID/data/BLOCK  # where the signal chunks of sensor SENSOR_ID should now be stored
ROOTDIR/SESSION/data/SENSOR_ID/info.json   # where info about the sensor should be stored (sample rate, etc.)
ROOTDIR/SESSION/info/                      # where the client's various files about the session will be stored
```

... again, and again, and again.

0.5.3 Need to put this on the cloud



And then one day...

We need to be in the clouds. Three use cases:

- Only on prem
- On prem, with immediate cloud availability
- Directly to the cloud

And in the cloud, we need to separate clients, and users within that...

Arg!

So...

```
s3://CLIENT_BUCKET/user_data/USER/sessions/SESSION/data/SENSOR_ID/data/BLOCK
etc.?
```

And now we're growing up, we need to involve some real databases.

Change is the only constant in life. (Heraclitus, a Greek philosopher.)

To name a few:

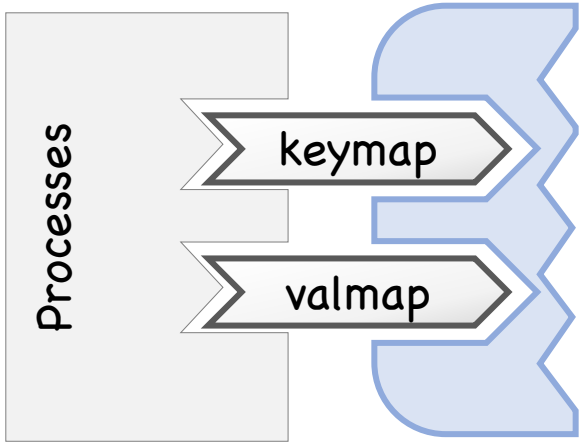
- Client wants us to use Azure, because they have a license for that already (sigh!).
- Clients already have their data, coming with all shapes and forms of files structures, audio file formats (even csv!!?!), which they give access to us through various means (ftp, dropbox link, google drive link, etc.)

You can't kill yourself by holding your breath (*)

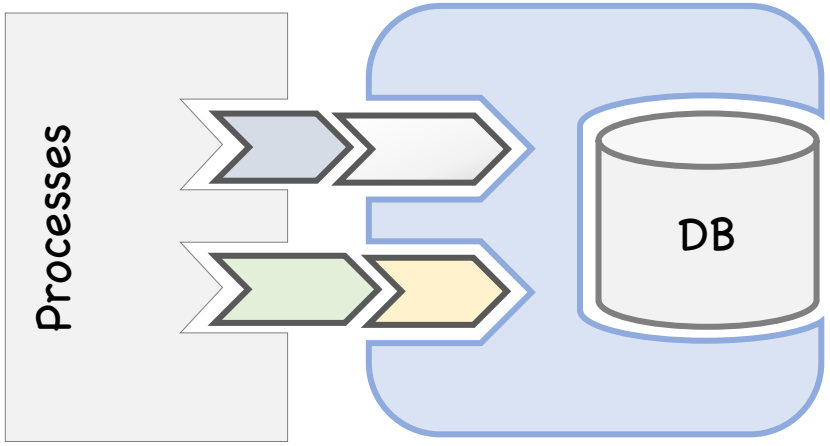
(*) https://biology.stackexchange.com/questions/73866/why-cant-we-kill-ourselves-by-holding-our-breath

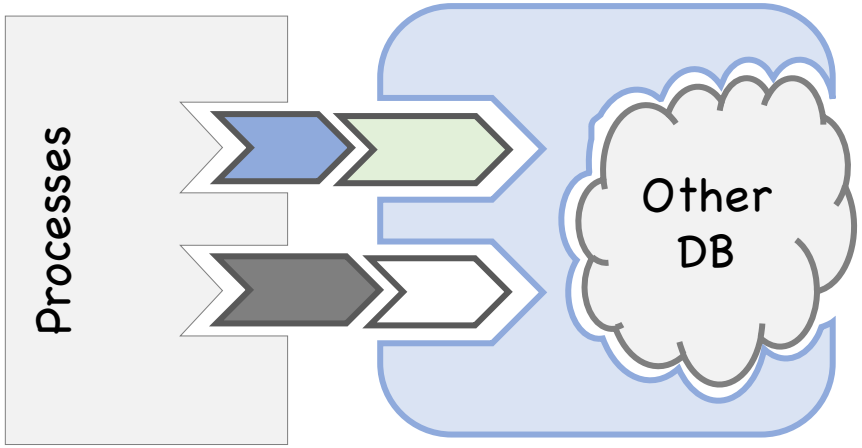
The solution? Nothing new: Just better design.

Code like this...



... so you can do this





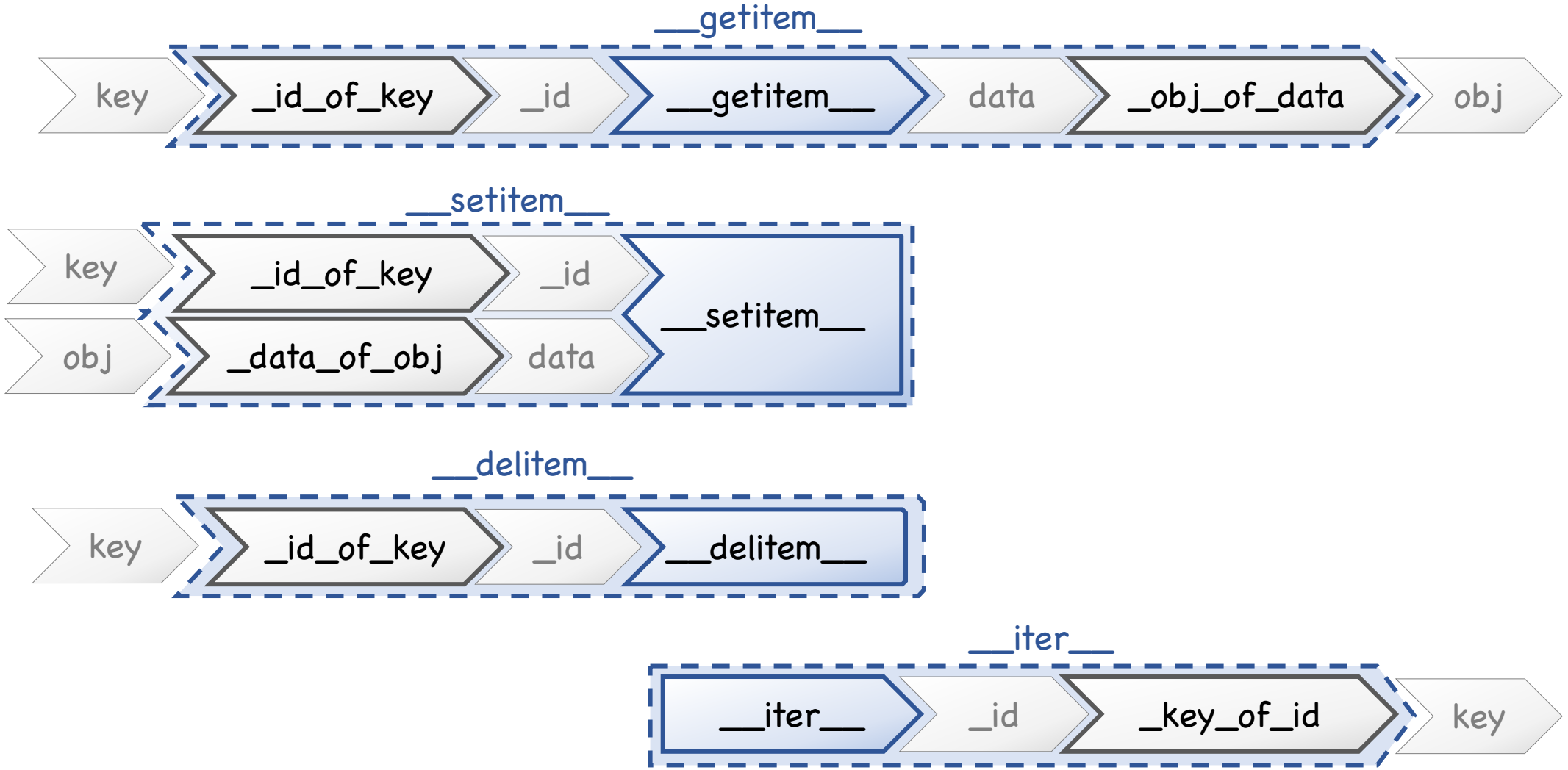
For those of you that ♡ acronyms:
It's not so much of an ORM as it is a DAO with a declarative SoC to be more DRY with storage.

What operations should we focus on first?
And what should we call those operations.

also known as...	Interface looks like...	dunder method
Store, write, create, dump	`d[k] = v`	`__setitem__(k, v)`
Retrieve, read, load	`v = d[k]`	`__getitem__(k)`
Delete, remove	`del(d[k])`	`__delitem__(k)`
List, get keys, ls	`list(d)` `for k in d: ...`	`__iter__()`

```
def __getitem__(self, k):
    return self._obj_of_data(self.persister.__getitem__(self._id_of_key(k)))

def __setitem__(self, k, v):
    return self.persister.__setitem__(self._id_of_key(k), self._data_of_obj(v))

def __delitem__(self, k):
    return self.persister.__delitem__(self._id_of_key(k))

def __iter__(self):
    return map(self._key_of_id, self.persister.__iter__())

def __len__(self):
    return self.persister.__len__()

def __contains__(self, k):
    return self.persister.__contains__(self._id_of_key(k))
```

Wrapping input/output keys/values of data accessors



So far py2store focuses on a dict-like interface to the storage concern

Your base **persister**: What happens PHYSICALLY when you store

Key Mapper: Indexing. The (2-way) mapping between the persister's key format and the format YOU want to (or need to) use

Val Mapper: Serialization. The (2-way) mapping between the persister's data format, and the object you get (you know, ORM stuff)

```
>>> persister = LocalFilePersister(root='/my/simpler/store')
>>>
>>> keymap = LinearNaming('{session}/data/{block:014.0f}.wav')
>>>
>>> valmap = AudioBytes(
...             sample_rate=44100, bit_depth=16, channels=1,
...             dtype='int16', format='WAV')
>>> s = mk_store(persister, keymap, valmap)
```

So now your code, no matter what the persister is, will look like this...

```
>>> len(s)
11
>>> list(s)[:3]
['/my/simpler/store/1566075321643/data/00000000000000.wav',
'/my/simpler/store/1566075321643/data/00000000172032.wav',
'/my/simpler/store/1566075321643/data/00000000086016.wav']
>>>
>>> k = list(s)[0]
>>> v = s[k]
>>>
>>> b = get_microphone_bytes(4096)
>>> s[4, 20] = b  # Store these bytes under (4,20)
>>>
>>> v = s['/my/simpler/store/4/data/00000000000042.wav']
>>> type(v)
<class 'numpy.ndarray'>
>>> len(v)
2048
>>> play_sound(v)
```

It's a matter of this:

```
filename = f"{int(session * 1000)}/{block:014.0f}"
filepath = os.path.join(storage_root, filename)
# make the necessary dirs!
dirpath = os.path.dirname(os.path.abspath(filepath))
os.makedirs(dirpath, exist_ok=True)
# save the data
with open(filepath, 'wb') as fp:
    fp.write(chk)
```

Versus that:

```
s[session, block] = chk
```

Or this:

```
filename = f"{int(session * 1000)}/{block:014.0f}"
filepath = os.path.join(storage_root, filename)
with open(filepath, 'rb') as fp:
    chk = fp.read()
wf, sr = sf.read(BytesIO(chk), samplerate=sr,
       channels=1, subtype='PCM_16', dtype='int16',
       format='RAW',  endian=None)
```

Versus that:

```
wf = s[session, block]
```

And when the time comes to store stuff somewhere else (S3, Mongo, SQL), with a different structure, or deal with a different data format or encoding...

Would you rather do this:

Or just modify a few things here?

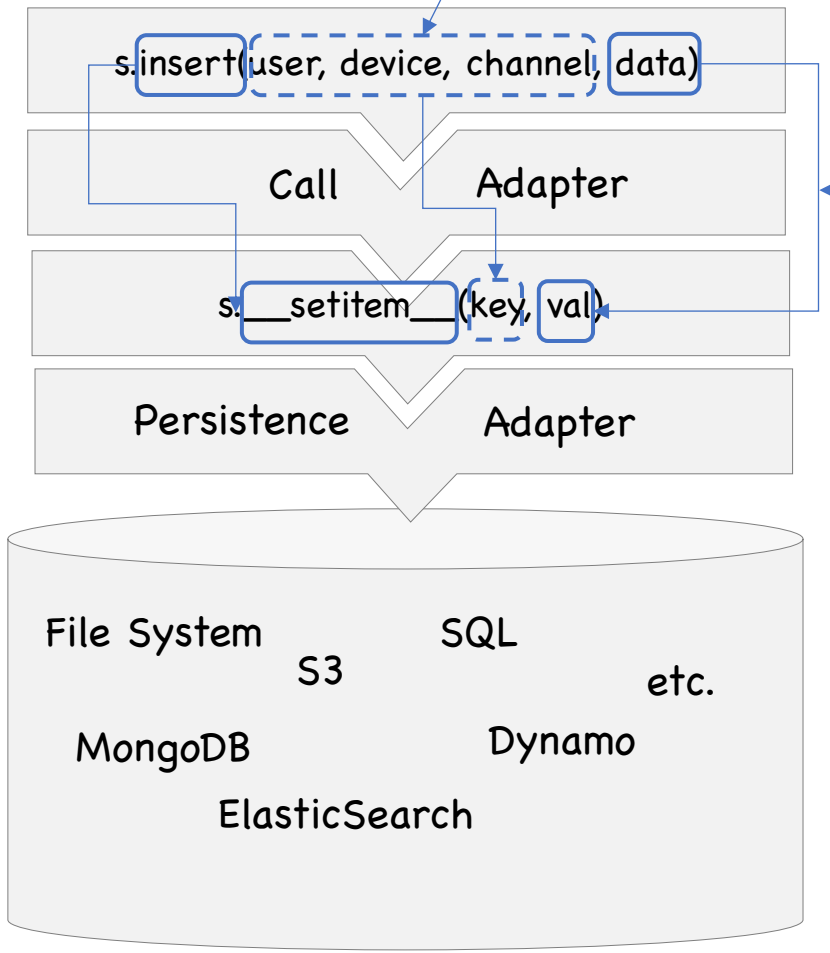
```
persister = LocalFilePersister(root='/my/simpler/store')

keymap = LinearNaming('{session}/data/{block:014.0f}.wav')

valmap = AudioBytes(
              sample_rate=44100, bit_depth=16, channels=1,
              dtype='int16', format='WAV')
s = mk_store(persister, keymap, valmap)
```

And not have to touch ANYTHING in the "business logic" code.

In one slide...

Say you want (or need) to use the word "insert" to write some data associated to a user/device/channel triple...

Adapter combines user, device, and channel to make a unique key for it



Adapter translates "insert" to it's internal write method name

Adapter serializes the data object to something that the persistence layer can actually persist

All adapter configurations are abstracted away so you can just write something somewhere and get on with more important stuff.

This is an example for writing, but the principle is the same for all other CRUD operations.

My email:	thor.whalen at analog (or gmail)
py2store repo:	https://github.com/i2mint/py2store
A demo notebook for this talk:	https://github.com/i2mint/examples/tree/master/pybay_2019