Introduction to Advanced Python

Charles-Axel Dein - June 2014, revised June 2016

License

- This presentation is shared under Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)
- More information about the license can be found here
- Please give proper attribution to the original author (Charles-Axel Dein)

Checkout my Github for
the source of this presentation
and more resources:

github.com/charlax/python-education

What's the goal of this presentation?

Goal 1:
Discover cool & underused Python features

Goal 2:
Improve your code's readability

Write code that explains to a human what we want the computer to do.

– D. Knuth

Do **not** use those patterns just when you feel like it

Readability can be achieved through conciseness...

... but it can also be hindered by overuse of magic

Let’s talk about a concrete example

Before

```
def toast(bread):
    if bread == "brioche":
        raise ValueError("Are you crazy?")

    toaster = Toaster()
    if toaster.has_bread("bagel"):
        raise ToasterNotEmptyError("Toaster already has some bagel")

    try:
        toaster.set_thermostat(5)

        for slot in [toaster.slot1, toaster.slot2]:
            slot.insert(bread)

    except BreadBurnedError:
        print("zut")

    finally:
        toaster.eject()
```

Let’s make it more expressive!

```python
def toast(bread):
    if bread == "brioche":
        raise ValueError("Are you crazy?")

    toaster = Toaster()
    if toaster.has_bread("bagel"):
        raise ToasterNotEmptyError(
            "Toaster already has some bagel")

    try:
        toaster.set_thermostat(5)

        for slot in [toaster.slot1, toaster.slot2]:
            slot.insert(bread)

    except BreadBurnedError:
        print("zut")

    finally:
        toaster.eject()
```

```python
@nobrioche
def toast(bread):
    toaster = Toaster()

    if "bagel" in toaster:
        raise ToasterNotEmptyError(
            "Toaster already has some bagel")

    with handle_burn():
        toaster.set_thermostat(5)

        for slot in toaster:
            slot.insert(bread)
```

It almost reads like
English.
(only with a French accent)

Note:

this is just an example!

In practice you'd want to be a bit more explicit about things.

Note:

examples were run with Python 3.4

Decorators

Attach additional responsibilities to an object

In Python , almost everything is an object

```
In [1]: def toast(): pass

In [2]: toast
Out[2]: <function __main__.toast>

In [3]: type(toast)
Out[3]: function
```

Python's decorators (@) are just syntactic sugar

```
def function():
    pass

function = decorate(function)

# is exactly equivalent to:


@decorate
def function():
    pass
```

Before

```
def toast(bread):
    if bread == 'brioche':
        raise ValueError("Are you crazy?")
    ...


def burn(bread):
    if bread == 'brioche':
        raise ValueError("Are you crazy?")
    ...
```

After

```
def no_brioche(function):
    def wrapped(bread):
        if bread == 'brioche':
            raise ValueError("Are you crazy?")
        return function(bread)
    return wrapped


@no_brioche
def toast(bread): pass


@no_brioche
def burn(bread): pass
```

Use cases

Caching

```
import functools


@functools.lru_cache(maxsize=32)
def get_pep(num):
    """Retrieve text of Python Enhancement Proposal."""
    resource = 'http://www.python.org/dev/peps/pep-%04d/' % num
    try:
        with urllib.request.urlopen(resource) as s:
            return s.read()
    except urllib.error.HTTPError:
        return 'Not Found'


pep = get_pep(8)
pep = get_pep(8)
```

Other use cases

- Annotating/registering (e.g. Flask's `app.route`)
- Wrapping in a context (e.g. mock's `mock.patch`)

To go further

- Parametrized decorators
- Class decorators
- ContextDecorator

Context manager

Originally meant to provide reusable
try... finally blocks

```python
with open("/etc/resolv.conf") as f:
    print(f.read())

# Is **roughly** equivalent to:

try:
    f = open("/etc/resolv.conf")
    print(f.read())
finally:
    f.close()
```

Context managers wrap execution in a context

```
class Timer(object):

    def __enter__(self):
        self.start = time.clock()
        return self

    def __exit__(self, exception_type, exception_value, traceback):
        self.end = time.clock()
        self.duration = self.end - self.start
        self.duration_ms = self.duration * 1000


with Timer() as timer:
    time.sleep(.1)

assert 0.01 < timer.duration_ms < 0.3
```

Before

```
try:
    os.remove('whatever.tmp')
except FileNotFoundError:
    pass
```

After

```
from contextlib import suppress

with suppress(FileNotFoundError):
    os.remove('somefile.tmp')
```

Use cases

redis pipeline

```
with redis_client.pipeline() as pipe:
    pipe.set('toaster:1', 'brioche')
    bread = pipe.get('toaster:2')
    pipe.set('toaster:3', bread)
```

SQL transaction

```python
from contextlib import contextmanager

@contextmanager
def session_scope():
    """Provide a transactional scope around a series of operations."""
    session = Session()
    try:
        yield session
        session.commit()
    except:
        session.rollback()
        raise
    finally:
        session.close()


def run_my_program():
    with session_scope() as session:
        ThingOne().go(session)
        ThingTwo().go(session)
```

Other use cases

- Acquiring/releasing a lock
- Mocking

To go further

- PEP 343
- contextlib module
  - @contextlib.contextmanager
  - contextlib.ExitStack
  - Single use, reusable and reentrant context managers

Iterator/generator

Iterators: what are they?

- An object that implements the iterator protocol
  - `__iter__()` returns the iterator (usually self)
  - `__next__()` returns the next item or raises StopIteration
- This method will be called for each iteration until it raises `StopIteration`

```
>>> import dis
>>> def over_list():
...     for i in None: pass
...
>>> dis.dis(over_list)
  2           0 SETUP_LOOP              14 (to 17)
              3 LOAD_CONST               0 (None)
              6 GET_ITER
        >>    7 FOR_ITER                 6 (to 16)
             10 STORE_FAST               0 (i)
             13 JUMP_ABSOLUTE            7
        >>   16 POP_BLOCK
        >>   17 LOAD_CONST               0 (None)
             20 RETURN_VALUE
```

```
>>> class Toasters(object):
...     """Loop through toasters."""
...     def __init__(self):
...         self.index = 0
...
...     def __next__(self):
...         resource = get_api_resource("/toasters/" + str(self.index))
...         self.index += 1
...         if resource:
...             return resource
...         raise StopIteration
...
...     def __iter__(self):
...         return self

>>> for resource in Toasters():
...     print(resource)
found /toasters/0
found /toasters/1
```

Generators

```
>>> def get_toasters():
...     while True:
...         resource = get_api_resource("/toasters/" + str(index))
...         if not resource:
...             break
...         yield resource
>>> for resource in get_resources():
...     print(resource)
found /toasters/0
found /toasters/1
```

Use cases of iterators

- Lazy evaluation of results one (batch) at time
  - Lower memory footprint
  - Compute just what you needed - can break in the middle (e.g. transparently page queries)
- Unbounded sets of results

To go further

- Other uses of yield (e.g. coroutines)
- Exception handling

Special methods

Special methods

- Method that starts and ends with “__”
- Allow operator overloading, in particular

Examples

- `__eq__`, `__lt__`, …: rich comparison (==, >…)
- `__len__`: called with len()
- `__add__`, `__sub__`, …: numeric types emulation (+, -…)
- Attribute lookup, assignment and deletion
- Evaluation, assignment and deletion of `self[key]`

Use case: operator overloading

```python
@functools.total_ordering
class Card(object):
    _order = (2, 3, 4, 5, 6, 7, 8, 9, 10, 'J', 'Q', 'K', 'A')
    def __init__(self, rank, suite):
        assert rank in self._order
        self.rank = rank
        self.suite = suite

    def __lt__(self, other):
        return self._order.index(self.rank) < self._order.index(other.rank)

    def __eq__(self, other):
        return self.rank == other.rank

ace_of_spades = Card('A', 'spades')
eight_of_hearts = Card(8, 'hearts')

assert ace_of_spades < eight_of_hearts
```

Why should they be used?

- Greater encapsulation of the logic,
  allowing objects to be manipulated without external function/methods
  - Allow uses of builtins that all Python developers know (`len`, `repr`, …)
  - Allow objects to be manipulated via operators (low cognitive burden)
- Allow use of certain keywords and Python features
  - `with` (`__enter__`, `__exit__`)
  - `in` (`__contains__`)

Before

```
if is_greater(ace_of_spades,
              eight_of_hearts):
    pass
```

```
if (ace_of_spades.rank
        > eight_of_hearts.rank):
    pass
```

After

```
if ace_of_spades > eight_of_hearts:
    pass
```

To go further

- Google “A Guide to Python's Magic Methods”
- Check how SQLAlchemy uses it to allow `session.query(Toaster.bread == ‘croissant’)`

Useful modules

collections

- namedtuple()
- Counter
- OrderedDict
- defaultdict

```
>>> # Find the ten most common words in Hamlet
>>> import re
>>> words = re.findall(r'\w+', open('hamlet.txt').read().lower())
>>> Counter(words).most_common(10)
[('the', 1143), ('and', 966), ('to', 762), ('of', 669), ('i', 631),
 ('you', 554),  ('a', 546), ('my', 514), ('hamlet', 471), ('in', 451)]
```

random

- randrange(start, stop[, step])
- randint(a, b)
- choice(seq)
- shuffle(x[, random])
- sample(population, k)

functools

- @lru_cache(maxsize=128, typed=False)
- partial(func, *args, **keywords)
- reduce(function, iterable[, initializer])

operator

- operator.attrgetter(attr)
- operator.itemgetter(item)

```
import operator as op


class User(object): pass


class Toaster(object): pass


toaster = Toaster()
toaster.slot, toaster.color = 1, 'red'
user = User()
user.toaster = toaster

f = op.attrgetter('toaster.slot', 'toaster.color')
assert f(user) == (1, 'red')
```

Profiling Python code

It’s extremely simple

```
import cProfile
import glob

cProfile.run("glob.glob('*')")
```

```
      164 function calls (161 primitive calls) in 0.000 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.000    0.000 <string>:1(<module>)
     1    0.000    0.000    0.000    0.000 fnmatch.py:38(_compile_pattern)
     1    0.000    0.000    0.000    0.000 fnmatch.py:48(filter)
     1    0.000    0.000    0.000    0.000 fnmatch.py:74(translate)
```

Conclusion

Other topics

- Memory allocation trace
- Metaclasses
- Descriptors and properties
- List, dict, set comprehensions
- Other modules: itertools, csv, argparse, operator, etc.

Thank you! Any questions?

Checkout my Github for
the source of this presentation
and more resources:

github.com/charlax/python-education