

# Python by smell

Advanced Python constructs for scientists

Pietro Berkes, Nagra Insight

# What's this smell?

- You're here because you started feeling the “code smell”
- Scientists in the wild tend to write this...
- What is the smell of the code in the notebooks?



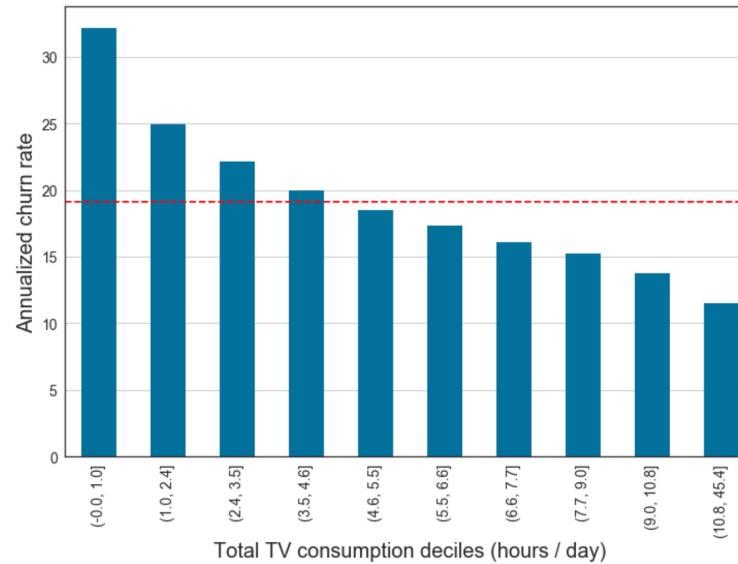
# What about this smell?

## 7.1 Total TV

```
In [118]: ttv_h_deciles = pd.qcut(ttv_h, 10)
ttv_deciles_churn = tv_merged.groupby(ttv_h_deciles).churned_all.mean() * 12 / 10 * 100
```

```
In [119]: with plt.rc_context(rc=get_style(figsize=(12 ,8))):
    ax = ttv_deciles_churn.plot.bar(color=blue)
    plt.grid(axis='y')
    plt.xlabel('Total TV consumption deciles (hours / day)')
    plt.ylabel('Annualized churn rate')
    plt.axhline(annual_tv_churn * 100, c='r', ls='--')

    t = ['{:.1f}, {:.1f}'.format(x.left, x.right) for x in ttv_deciles_churn.index]
    plt.xticks(range(len(t)), t)
```



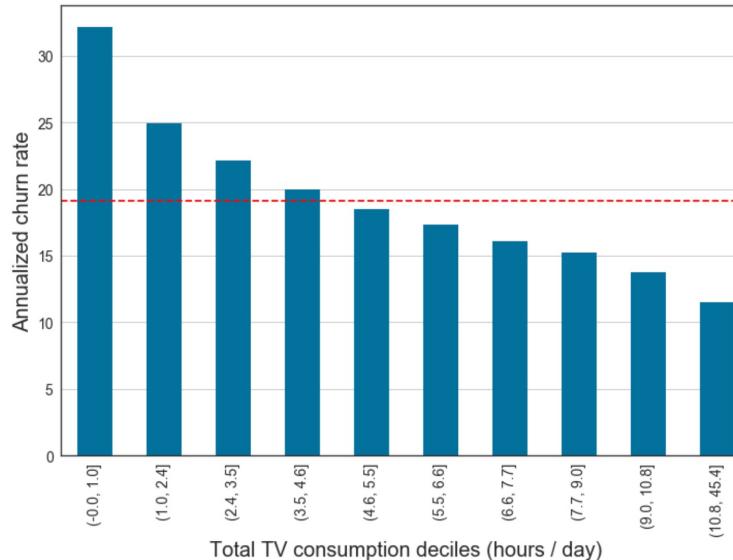
# What about this smell?

## 7.1 Total TV

```
In [118]: ttv_h_deciles = pd.qcut(ttv_h, 10)
ttv_deciles_churn = tv_merged.groupby(ttv_h_deciles).churned_all.mean() * 12 / 10 * 100
```

```
In [119]: with plt.rc_context(rc=get_style(figsize=(12 ,8))):
    ax = ttv_deciles_churn.plot.bar(color=blue)
    plt.grid(axis='y')
    plt.xlabel('Total TV consumption deciles (hours / day)')
    plt.ylabel('Annualized churn rate')
    plt.axhline(annual_tv_churn * 100, c='r', ls='--')

    t = ['{:.1f}, {:.1f}'.format(x.left, x.right) for x in ttv_deciles_churn.index]
    plt.xticks(range(len(t)), t)
```



## 7.2 Playback

```
In [120]: replay_deciles = pd.qcut(tv_merged.PLAYBACK / ttv_sec_to_hpd, 10)
replay_deciles_churn = tv_merged.groupby(replay_deciles).churned_all.mean() * 12 / 10 * 100

In [121]: with plt.rc_context(rc=get_style(figsize=(12 ,8))):
    ax = replay_deciles_churn.plot.bar(color=blue)
    plt.grid(axis='y')
    plt.xlabel('Replay TV consumption deciles (hours / day)')
    plt.ylabel('Annualized churn rate')
    plt.axhline(annual_tv_churn * 100, c='r', ls='--')

    t = ['{:.1f}, {:.1f}'.format(x.left, x.right) for x in replay_deciles_churn.index]
    plt.xticks(range(len(t)), t)
```

```
In [122]: replay_h = pd.cut(tv_merged.PLAYBACK / ttv_sec_to_hpd, np.arange(0, 8), include_lowest=True)
replay_churn = tv_merged.groupby(replay_h).churned_all.mean() * 12 / 10 * 100
```

```
In [123]: with plt.rc_context(rc=get_style(figsize=(12 ,8))):
    ax = replay_churn.plot.bar(color=blue)
    plt.grid(axis='y')
    plt.xlabel('Replay TV consumption (hours / day)')
    plt.ylabel('Annualized churn rate')
    plt.axhline(annual_tv_churn * 100, c='r', ls='--')
```

## 7.3 Trends

```
In [137]: tv_delta = pd.cut(tv_merged.TTV_201703_delta, np.arange(-5.5, 5.6, 1.0))
tv_delta_churn = tv_merged.groupby(tv_delta).churned_all.mean() * 12 / 10 * 100
```

```
In [138]: with plt.rc_context(rc=get_style(figsize=(12 ,8))):
    ax = tv_delta_churn.plot.bar(color=blue)
    plt.grid(axis='y')
    plt.xlabel('TV consumption trend')
    plt.ylabel('Annualized churn rate')
    plt.axhline(annual_tv_churn * 100, c='r', ls='--')
```

```
In [1651]: tv_delta = pd.cut(tv_merged.TTV_201703_delta, np.arange(-5.5, 5.6, 1.0))
tv_delta_churn_some = tv_merged[at_least_some_ttv_mask].groupby(tv_delta[at_least_some_ttv_ma
```

```
In [1652]: with plt.rc_context(rc=get_style(figsize=(12 ,8))):
    ax = tv_delta_churn_some.plot.bar(color=blue)
    plt.grid(axis='y')
    plt.xlabel('TV consumption trend')
    plt.ylabel('Annualized TV+Vodafone churn rate')
    plt.axhline(annual_tv_churn * 100, c='r', ls='--')
```

# What is wrong with smelly code?

- **Redundant, not flexible:** an update in one place would need to be duplicated everywhere
- **Hard to read and test:** the code that performs the interesting computation is mixed with the code that does the repetitive boilerplate



# Objective

- This is a code smell detection crash course for scientific programming
- **Advanced Python constructs are the way to get rid of the smell!**



# All the “advanced Python constructs” smells

## The smell of generators

```
for ... :  
    # Transform / filter  
    for ... :  
        # Transform / filter  
        # The interesting part of the code
```

## The smell of classes

```
def first_function(x, y, z):  
    # Something  
  
def second_function(x, y, z):  
    # Something else  
  
def third_function(x, y, z):  
    # Something more
```

## The smell of context managers

```
# Prepare  
try:  
    # The code you care about  
finally:  
    # Clean up
```

## The smell of decorators

```
def my_function(x, y, z):  
    # Common boilerplate at beginning  
    # Function-specific part  
    # Common boilerplate at end
```

# The smell of context managers

Before executing the code,  
something needs to happen:  
open a file, connect to a DB,  
initialize some hardware

Once the code has  
executed, we need to clean  
up, **even if an error  
occurred**: close the file,  
commit / revert SQL  
transactions, disconnect  
from hardware

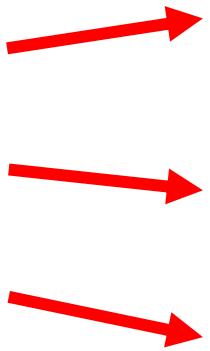
```
# Initialize context
try:
    # The code you care about
finally:
    # Clean up context
```

... becomes ...

```
with my_context_manager():
    # The code you care about
```

# The smell of classes

The same set of parameters is needed for a set of functions.  
In code calling this function, one needs extra code to keep these parameters in sync.



```
def first_function(x, y, z):  
    # Something  
  
def second_function(x, y, z):  
    # Something else  
  
def third_function(x, y, z):  
    # Something more
```

... becomes ...

```
class Xyz:  
    def __init__(self, x, y, z):  
        ...
```

```
def first_function(xyz):  
    # Something  
  
def second_function(xyz):  
    # Something else  
  
def third_function(xyz):  
    # Something more
```

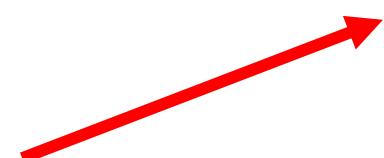
# The smell of generators

One or more nested loops,  
variables over which one  
iterates requires some extra  
transformation or filtering



```
for ... :  
    # Transform / filter  
    for ... :  
        # Transform / filter  
        # The interesting part of the code
```

This is the part that  
actually does some  
interesting computation. At  
the moment, it's hard to  
test it!



... becomes ...

```
for ... in my_generator():  
    # The interesting part of the code
```

# The smell of decorators

Boilerplate code at the start  
and/or end of functions.

Typical cases: logging,  
deprecation, conditions  
checks, caching

```
def my_function(x, y, z):  
    # Boilerplate at start  
    # Function-specific part  
    # Boilerplate at end
```

This is the part that is  
specific to what the  
function actually does

... becomes ...

```
@my_decorator  
def my_function(x, y, z):  
    # Function-specific part
```

# One smell at the time: Context managers

# The smell of context managers

Before executing the code,  
something needs to happen:  
open a file, connect to a DB,  
initialize some hardware

Once the code has  
executed, we need to clean  
up, **even if an error  
occurred**: close the file,  
commit / revert SQL  
transactions, disconnect  
from hardware

```
# Initialize context
try:
    # The code you care about
finally:
    # Clean up context
```

... becomes ...

```
with my_context_manager():
    # The code you care about
```

# Go to “context managers” notebooks

# Recap: Context managers

- Context managers eliminate the smell of repeatedly setting up and cleaning up an environment in which code needs to run

```
from contextlib import contextmanager

@contextmanager
def my_context(params):
    print('Set up environment')
    try:
        yield # Here the block of code is executed
    finally:
        print('Clean up environment')

with my_context(params):
    print('Do something interesting here')
```

# FYI: most general way of defining context manger

- `@contextmanager` is a shortcut for writing a class with magic methods `__enter__` and `__exit__`:

```
class MyContext():

    def __init__(self, params):
        self.params = params

    def __enter__(self):
        print('Set up environment')

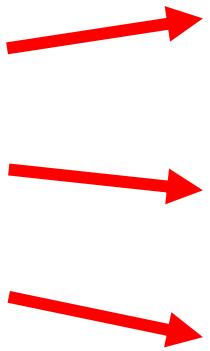
    def __exit__(self, *args):
        # This is called even if there is an exception!
        print('Clean up environment')


with MyContext(params):
    print('Do something interesting here')
```

# One smell at the time: classes (an introduction)

# The smell of classes

The same set of parameters is needed for a set of functions.  
In code calling this function, one needs extra code to keep these parameters in sync.



```
def first_function(x, y, z):  
    # Something  
  
def second_function(x, y, z):  
    # Something else  
  
def third_function(x, y, z):  
    # Something more
```

... becomes ...

```
class Xyz:  
    def __init__(self, x, y, z):  
        ...
```

```
def first_function(xyz):  
    # Something  
  
def second_function(xyz):  
    # Something else  
  
def third_function(xyz):  
    # Something more
```

Go to “classes” notebooks

# Recap: Classes

- Classes are used to get rid of set of parameters that belong together and are passed over and over to a set of functions
- Classes are templates for bundles of data and “methods”, i.e. functions that have access to the data stored in an instance
- A “class method” is used to build an instance in some alternative way, e.g. using data from a file

# Recap: Class structure

```
class MyClass:  
    def __init__(self, param1, param2):  
        self.param1 = param1  
        if param2 is None:  
            param2 = 12.3  
        self.param2 = param2
```

The constructor is used to first populate an instance, called by convention “self”

```
def my_method(self, foo, bar):  
    result = self.param2 * foo + self.param2 * bar  
    return result
```

Classes can define “methods”, i.e. functions that have access to the data stored in an instance

```
def to_json(self):  
    params = {  
        'param1': self.param1,  
        'param2': self.param2,  
    }  
    return json.dumps(params)
```

```
@classmethod  
def from_json(cls, json_str):  
    json_dict = json.loads(json_str)  
    instance = cls(json_dict['params1'], json_dict['params2'])  
    return instance
```

A “class method” can be used to build an instance in some alternative way, e.g. using data from a file

```
instance = MyClass(5, 18)  
instance = MyClass.from_json(json_str)
```

Here is how you create instances from the constructor or a class method

# What belongs to a class?

- YES
  - + Data that always belongs together: better create several simple classes than one class that contains everything
  - + Methods to load/save data, create data bundle in different ways (factory methods)
  - + Methods to update parameters
- NO
  - Methods to visualize data: follow the Model-View pattern.  
You will want to visualize the data in many different ways, better have separate utility visualization functions that take one of the instances as input and visualize them.
  - Similarly, anything for which you can imagine to write 5 different variants depending on your mood

# Another smell of classes

Several “specializations” of conceptually similar functions



```
for data in list_of_data:  
    if data['type'] == 'TYPE1':  
        type1_foo(data)  
        type1_bar(data)  
    elif data['type'] == 'TYPE2':  
        type2_foo(data)  
        type2_bar(data)
```

... becomes ...

Classes may have methods with the same interface, the class type determines the “specialization”.



```
for instance in instances:  
    instance.foo(data)  
    instance.bar(data)
```

One can even define a hierarchy of classes where some methods are re-used!

# Another smell of classes – simple example

Several “specializations” of conceptually similar functions



```
for data in geometric_objects:  
    if data['type'] == 'SQUARE':  
        area = area_square(data)  
    elif data['type'] == 'CIRCLE':  
        area = area_circle(data)
```

Classes may have methods with the same interface, the class type determines the “specialization”.



```
for instance in geometric_objects:  
    area = instance.area()
```

One can even define a hierarchy of classes where some methods are re-used!

... becomes ...

# Real example: sklearn

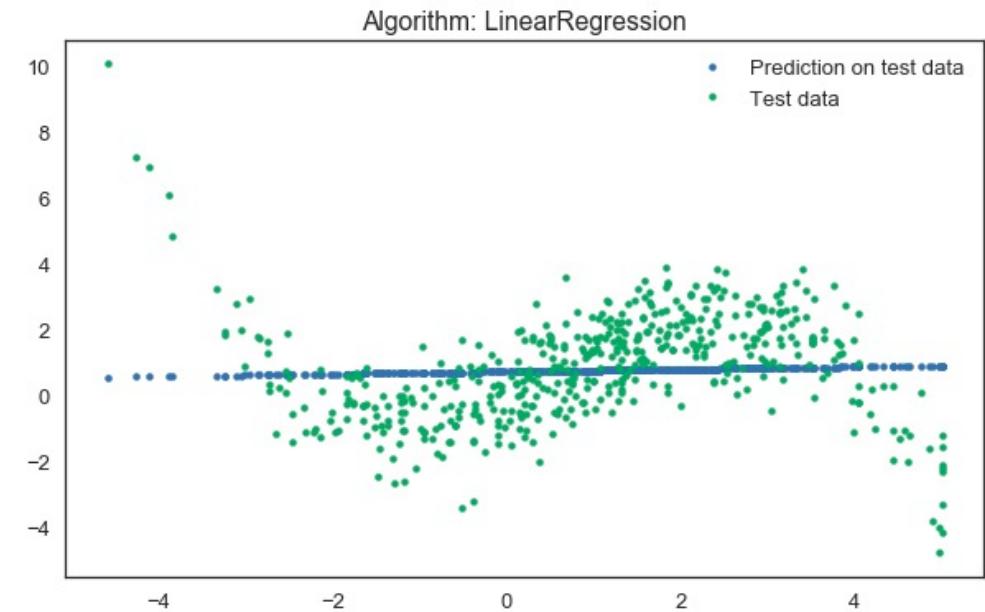
```
from sklearn.linear_model import LinearRegression

model = LinearRegression()
model.fit(X_train, y_train)

score = model.score(X_test, y_test)
print('Model score:', round(score, 4))

y_pred = model.predict(X_test)
with plt.rc_context(rc={'figure.figsize': (10, 6)}):
    plt.plot(X_test, y_pred, '.', label='Prediction on test data')
    plt.plot(X_test, y_test, '.', label='Test data')
    plt.legend()
    plt.title('Algorithm: ' + model.__class__.__name__)
```

Model score: 0.0066



# Real example: sklearn

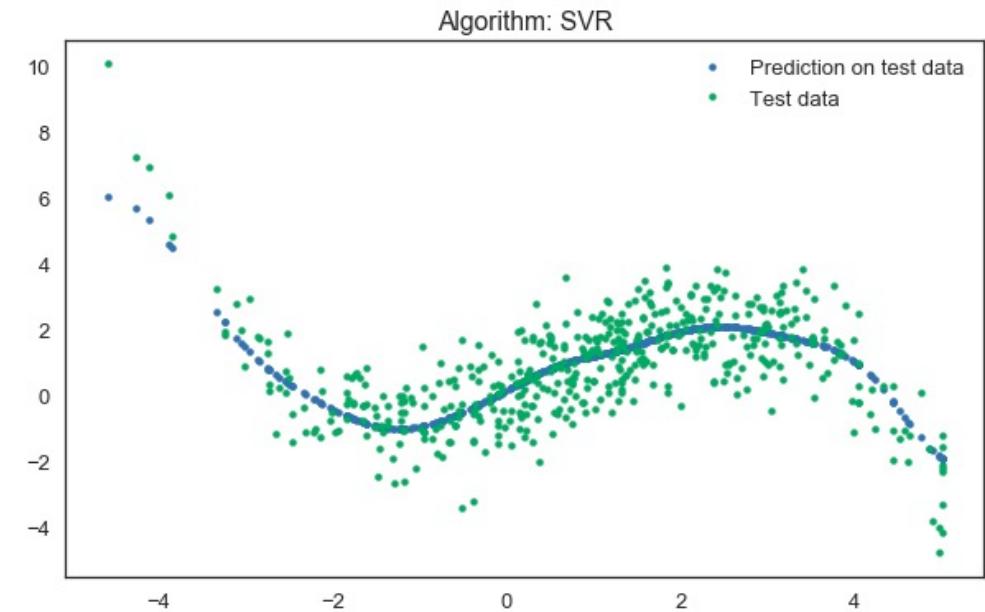
```
from sklearn.svm import SVR

model = SVR(kernel='rbf')
model.fit(X_train, y_train)

score = model.score(X_test, y_test)
print('Model score:', round(score, 4))

y_pred = model.predict(X_test)
with plt.rc_context(rc={'figure.figsize': (10, 6)}):
    plt.plot(X_test, y_pred, '.', label='Prediction on test data')
    plt.plot(X_test, y_test, '.', label='Test data')
    plt.legend()
    plt.title('Algorithm: ' + model.__class__.__name__)
```

Model score: 0.6144



# Keep your nose ready!

## The smell of generators

```
for ... :  
    # Transform / filter  
    for ... :  
        # Transform / filter  
    # The interesting part of the code
```

## The smell of classes

```
def first_function(x, y, z):  
    # Something  
  
def second_function(x, y, z):  
    # Something else  
  
def third_function(x, y, z):  
    # Something more
```

## The smell of context managers

```
# Prepare  
try:  
    # The code you care about  
finally:  
    # Clean up
```

## The smell of decorators

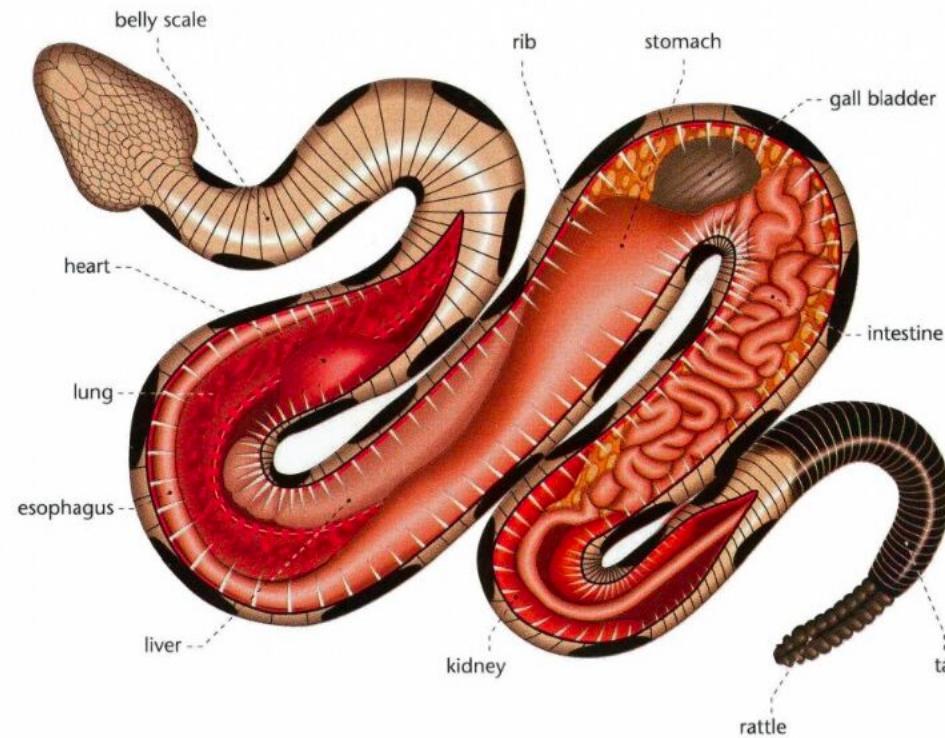
```
def my_function(x, y, z):  
    # Common boilerplate at beginning  
    # Function-specific part  
    # Common boilerplate at end
```

# Where to go from here...



[realpython.com](https://realpython.com)

# Thank you!





# One smell at the time: generators first

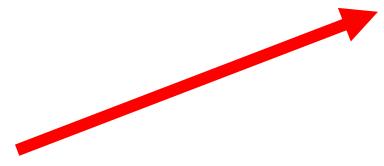
# The smell of generators

One or more nested loops,  
variables over which one  
iterates requires some extra  
transformation or filtering



```
for ... :  
    # Transform / filter  
    for ... :  
        # Transform / filter  
        # The interesting part of the code
```

This is the part that  
actually does some  
interesting computation. At  
the moment, it's hard to  
test it!



... becomes ...

```
for ... in my_generator():  
    # The interesting part of the code
```

# Example

```
comment_prefixes = ['# ', '-- ', 'REM ']

filename1 = 'first_commented_data.csv'
print('Load data from', filename1)
with open(filename1, 'rt') as f:
    valid_lines = []
    for line in f:
        for prefix in comment_prefixes:
            if line.startswith(prefix):
                break
    else:
        data = [int(x) for x in line.split(',')]
        valid_lines.append(data)

data1 = pd.DataFrame(valid_lines, columns=['unci', 'dunci', 'trinci', 'quari'])
```

Ideally...

```
parse_data(filter_comments(readfiles(filenames)))
```

This is actually what we'd like to write:

1. Extract each for-loop step in a reusable element and give it a nice name
2. Do all of this without actually loading all the data in memory for the intermediate steps!

Let's get there step by step...

# Go to “generators” notebooks

# Recap: Generators

- Generators are used to get rid of repetitive loops, often nested "for" loops followed by filtering of the data
- Generators return one item at the time, the list of items does not need to be in memory
- A generator is defined as a function containing the keyword "yield":

```
def odd_numbers(n):
    """ Generator for the first `n` odd numbers. """
    for i in range(n):
        # Use `yield` instead of `return`: execution will start again from here
        yield i * 2 + 1

for i in odd_numbers(5):
    print(i)
```