

*Intermediate!*

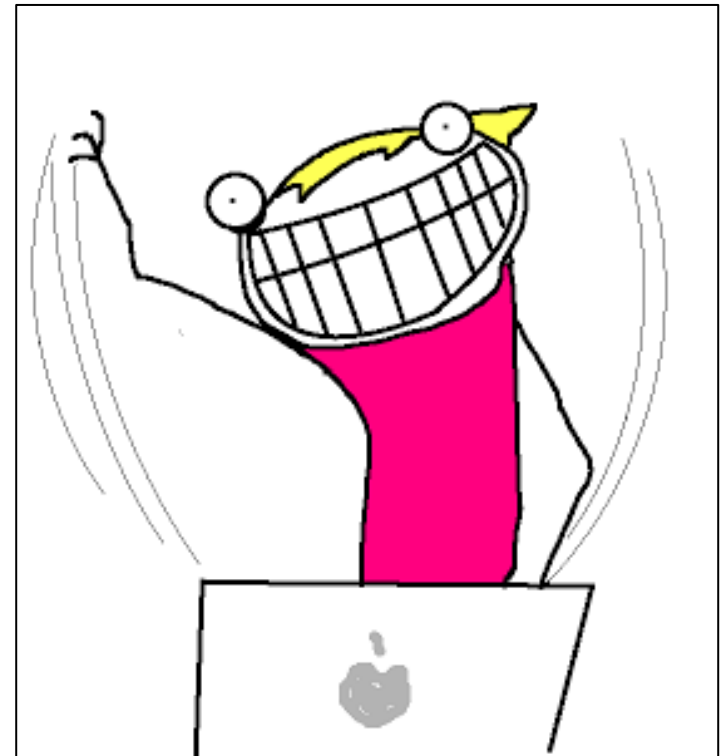


python

Caroline Harbitz  
January 28, 2016

# Why am I giving this talk?

- I went to PyCon 2015 and saw all this cool—but confusing—code.
- To help YOU!
  - Get excited
  - Be more efficient
  - Understand others' code



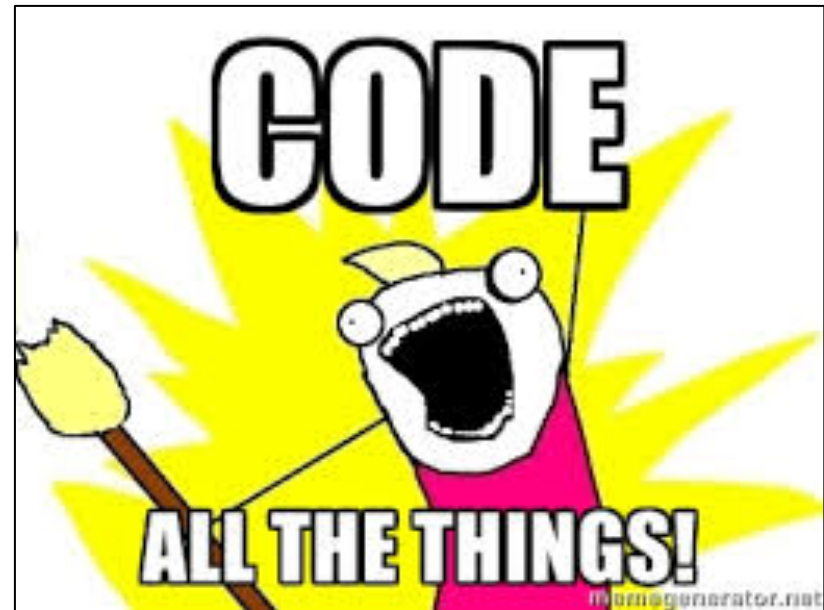
# Format

## Part 1

Short lectures about:

- List comprehensions
- Iterators
- Generators
- Decorators

## Part 2



...that you want to.

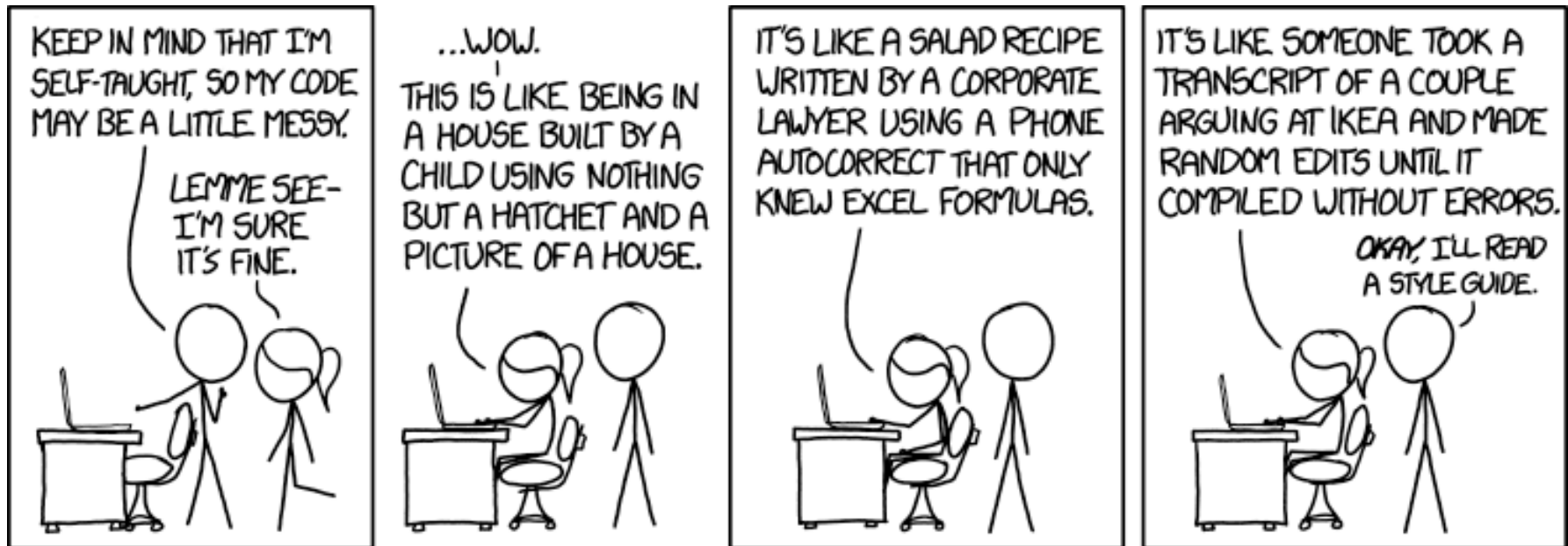
# The Zen of Python

Beautiful is better than ugly.



# The Zen of Python

Readability counts.



# List comprehension

Anatomy of a list comprehension:

```
new list = [transform iterate filter]
```

# How to use list comprehension (LC)

- Say you want to print the square of the odd numbers in a list :

```
some_list = [1, 2, 3, 4, 5]
```

- The end result will be: [1, 9, 25]
- How can we do this using a list comprehension?

# How to use LC

1. First things first:

```
[]
```

2. Iterate over the sequence:

```
[for num in my_list]
```

3. Write the filter condition:

```
[for num in my_list if num % 2 == 1]
```

4. Include the transformed result:

```
[num**2 for num in my_list if num % 2 == 1]
```

5. Optional: save result to a new list

```
squared_ints = [num**2 for num in my_list \  
                 if num % 2 == 1]
```



# When to use LC

- When you're using a loop to transform a sequence
- When you don't want to write code like this:

```
numbers = [0,1,2,3,4,5,6,7,8,9]
size = len(numbers)
i = 0
evens = []
while i < size:
    if i % 2 == 0:
        evens.append(i)
    i += 1
```

# When NOT to use LC

Don't use list comprehensions for:

- deeply nested iterations,
- complicated transformations, or
- code that would be easier understood if it were written using **for** or **while** loops.

# Iterators

**Iterator:** An object that implements the iterator protocol.

Iterators must implement these two methods:

1. `__iter__()`
2. `next()`

**Iterable object:** an object that can yield objects one at a time.

# Iterate? Iterable? Iterator?

- To **iterate**: given a collection of values, take one value after the other from the collection.
- An **iterable** is an object that you can get an iterator from either:
  1. An `__iter__()` method
  2. A `__getitem__()` method
- An **iterator** is an object from which you can get one value at a time with **next()**.

# You've seen this before!

Looping over...

- lists →
- strings
- dictionary keys →
- files

```
for n in [5,6,7,8,9]:  
    print n
```

```
for key in {"x": 1, "y": 2}:  
    print key
```



Iterables

# How to use iterators

```
my_iter = iter(collection)
```



has a `__next__()` method

# How to use iterators

```
>>> s = 'abc'
```

```
>>> it = iter(s)
```

```
>>> it
```

```
<iterator object at 0x1014b6110>
```

```
>>> next(it)
```

```
'a'
```

```
>>> next(it)
```

```
'b'
```

# How to use iterators

```
>>> next(it)
```

```
'c'
```

```
>>> next(it)
```

```
StopIteration
```



# When to use iterators

You already do!



# Generators

- Generators are functions that use **yield** expressions.
- When called, generators immediately return an **iterator**.
- Using **next()**, the iterator advances the generator to its next yield expression.

## First: a regular function

```
def firstn(n):  
    num, nums = 0, []  
    while num < n:  
        nums.append(num)  
        num += 1  
    return nums  
  
sum_of_first_n = sum(firstn(1000000))
```

# How to use generators

```
def firstn(n):  
    num = 0  
    while num < n:  
        yield num  
        num += 1  
  
sum_of_first_n = sum(firstn(1000000))
```

# Generator expressions

- A generalization of list comprehensions and generators
- Yield one item at a time

# Generator expressions (lazy!)

Syntax:

```
lc_doubles = [2 * n for n in [1,2,3,4,5]]
```



```
genexp = (2 * n for n in [1,2,3,4,5])  
genexp_doubles = list(genexp)
```

# Materialize



```
genexp = (2 * n for n in range(1,6))  
genexp_doubles = list(genexp)
```

- Providing the generator expression as an argument to **list()** builds the entire list.
- Use **range()** or **xrange()** to create sequences of numbers.
- Other built-in functions that take iterables:
  - `sorted()`
  - `min()`, `max()`
  - `sum()`
  - `dict()`
  - `all()`, `any()`

# Generator expression example

```
>>> gen = (value for value in [4,5,6,7,8,9]\n            if value > 5)
```

```
>>> gen
```

```
<generator object <genexpr> at 0x103bb6d70>
```

```
>>> next(gen)
```

```
6
```

```
>>> min(gen)
```

```
7
```

```
>>> min(gen)
```

```
ValueError: min() arg is an empty sequence
```



# Equivalent functions

This generator:

```
def pos_generator(seq):  
    for x in seq:  
        if x >= 0:  
            yield x
```

Is equivalent to this generator expression:

```
def pos_gen_exp(seq):  
    return (x for x in seq if x >= 0)
```

And they both produce the same result:

```
>>> list(pos_generator(range(-5, 5))) == \  
      list(pos_gen_exp(range(-5, 5)))  
True
```

# When to use generators

- You have a lot of data to iterate over
- To avoid materialization

# When to NOT use generators

- Slicing is necessary
- They can be tricky to debug
  - Can only access values one at a time, not the whole collection

# Decorators

- Do something before, during, and/or after some code
- Goal: reduce boilerplate code
- Extend the behavior of a function without modifying it
- “Design pattern that allows behavior to be added to an existing object dynamically.”

# Functions: review

```
def foo():  
    """Docstring"""  
    print 'Hello!'
```

```
>>> foo  
>>> foo()
```

```
>>> bar = foo  
>>> bar.__name__  
'foo'
```

Parameters: positional,  
keyword, variable (\*args),  
variable keyword (\*\*kwargs)

```
def get_foo():  
    return foo
```

```
>>> dir(foo)
```

```
def adder():  
    def add(x,y):  
        return x + y  
    return add
```

```
>>> adder()  
>>> adder()(2,4)
```

# Generic decorator pattern

```
def mydecorator(some_function):  
    def inner_function(*args, **kw):  
        # do some stuff before  
        result = some_function(*args,  
                                *kw)  
        # do some stuff after  
        return result  
    return inner_function
```

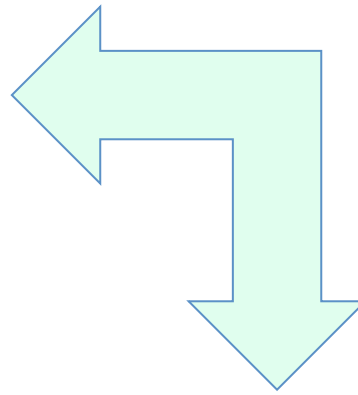
To use:

```
@mydecorator  
def some_function():  
    . . .
```



# Equivalent syntax

```
@mydecorator  
def myfunc():  
    pass
```



```
def myfunc():  
    pass  
myfunc = mydecorator(myfunc)
```

```
def verbose(my_func):  
    def inner_function(*args, **kwargs):  
        print "before", my_func.__name__  
        result = my_func(*args, **kwargs)  
        print "after", my_func.__name__  
        return result  
    return inner_function
```

```
@verbose  
def print_message():  
    print "Hello there!"
```

```
>>> print_message()  
before print_message  
Hello there!  
after print_message
```



# Flask example

```
from app import app

@app.route('/')
@app.route('/index')
def index():
    return "Hello, World!"
```

<http://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world>

# When to use decorators

- Minimize boilerplate code and simplify functions
- Logging
- Error handling
- Caching expensive calculations
- Retrying functions that might fail

# Playtime!

\*Suggested\* order:

1. List comprehensions
2. Iterators
3. Generators
4. Decorators

Github repository:

[github.com/cterp/pyladies-intermediate-python](https://github.com/cterp/pyladies-intermediate-python)

# Reading list

1. <https://docs.python.org/>
2. Slatkin, Brett: Effective Python: 59 Specific Ways to Write Better Python. Addison-Wedley, 2015.
3. Alchin, Marty: Pro Python: Advanced coding techniques and tools. Apress, 2010.
4. Anything Matt Harrison writes about Python.

# What to do next

Module suggestions:

- iterator
- collections
  - Tired of counting?
- itertools



This workshop was really only about  
one thing...

Lazily materialize objects whenever possible.

