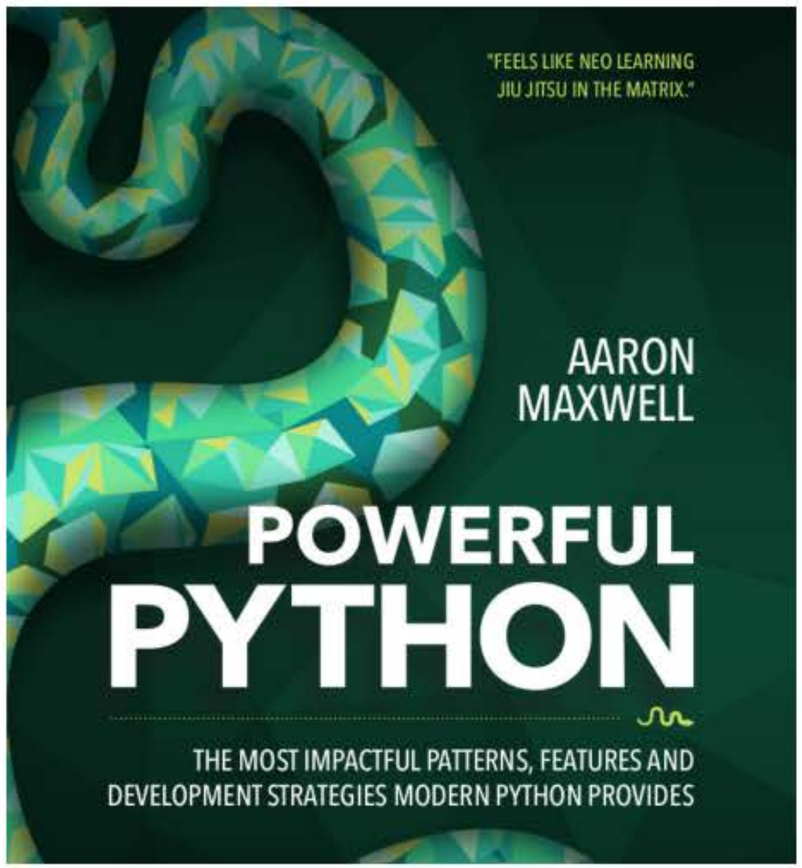
Python: Beyond the Basics

Welcome

I'm your host, Aaron Maxwell.

- Author of **Powerful Python**
- aaron@powerfulpython.com
- On twitter:
  - @powerfulpython (professional)
  - @redsymbol (personal)



Our focus in this class: **Responsive, Scalable, and Maintainable Code Patterns.** Fully participating will **magnify your ability** to expressively write **powerfully effective** Python code.

Broad Itinerary

Today:

- Generators for scalable, responsive code
- Generator Design Patterns for Scalable Composability
- Understanding iterators, and Python's iterator protocol
- List comprehensions for rich, expressive data structures
- And... Homework!

Tomorrow:

- Comprehensions Part 2, and how they relate to Generators
- Quick review of Python's object syntax (~ 15 minutes)
- Properties For Clean Design and Refactoring
- Object design patterns
- And... More homework!

How we will proceed

Download courseware ZIP:

- Click on the green "Resource List" icon
- Or fetch from: https://powerfulpython.com/courseware-btb.zip

What's included:

- PDF course book
- Slides
- README.txt with pointers
- Labs (i.e., programming exercises - more on that later)

Give you a break every hour (10 minutes).

Give me a thumbs up. (Let's try it now)

Ask questions anytime.

Python versions

Most code I show you will run in both Python 2 and 3.

Where it's different, I'll code in Python 3, and point out the differences. (There won't be many.)

You can do the programming exercises in either 3, or 2.7.

What makes perfect?

Practice, practice, practice.

- Practice syntax (typing things in)
- Practice programming (higher-level labs)

I expect you to do your part!

You **exponentially** get out of this what you put into it.

GO FOR IT.

Running the labs

Labs are the main programming exercises. You are given a failing automated test; your job is to write Python code to make it pass.

Simply run it as a Python program, any way you like. (For example, "python3 helloworld.py")

Run unmodified first, so you can see the failure report.

When done, click the thumb's up, and find someone to high-five.

Then: Move on to the extra credit.

Lab: helloworld.py

Let's do our first lab now: 'helloworld.py`

- In `labs/py3` for Python 3.x, or `labs/py2` for 2.7

When you finish:

- Give me a HIGH FIVE! in the chat room, and click Thumbs up, so I know you're done.
- Proceed to `helloworld_extra.py`

You'll know the tests pass when you see:

```
*** ALL TESTS PASS ***
Give someone a HIGH FIVE!
```

When that's done: Open and skim through **PythonBeyondTheBasics.pdf** - just notice what topics interest you.

Getting the most

We'll take some class time for each lab. You may not finish, but it's **critically important** that you at least start when I tell you to.

After we're done for the day, find time to finish all the main labs before tomorrow.

Solutions are provided. Use them wisely, not foolishly:

- After you get the lab passing, compare your code to the official solution.
- Other than that, don't look at them if you can avoid it.
- The more work you can do on your own, the more you will learn. Peek at the solution to get a hint when you really need it.

Optional (**only** for future master Pythonistas): Do all the extra labs as well, as soon as you can manage.

Generators in Python

Square Processing

Let's say you need to do something with square numbers.

```
def fetch_squares(max_root):
    squares = []
    for x in range(max_root):
        squares.append(x**2)
    return squares

MAX = 5
for square in fetch_squares(MAX):
    do_something_with(square)
```

This works. But...

Maximum MAX

What if MAX is not 5, but 10,000,000? Or 10,000,000,000? Or more?

What if you aren't doing arithmetic to get each element, but making a truly expensive calculation? Or making an API call? Or reading from a database?

Now your program has to wait... to create and populate a huge list...
before the second for-loop can even START.

Lazily Looping

The solution is to create an iterator to start with, which lazily computes each value just as it's needed. Then each cycle through the loop happens just in time.

The Iterator Protocol

Here's how you do it in Python:

```
class Squares:
    def __init__(self, max_root):
        self.max_root = max_root
        self.root = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.root == self.max_root:
            raise StopIteration
        value = self.root ** 2
        self.root += 1
        return value

for square in Squares(5):
    print(square)
```

There's got to be a better way

Good news. There's a better way.

It's called the **generator**. You're going to love it!

- Sidesteps potential memory bottlenecks, to greatly improve scalability and performance
- Improves real-time responsiveness of the application
- Can be chained together in clear, composable code patterns for better readability and easier code reuse
- Provides unique, valuable mechanisms of encapsulation. Concisely expressive and powerfully effective coding
- A key building block of the async services in Python 3

Yield for Awesomeness

A generator looks just like a regular function, except it uses the `yield` keyword instead of `return`.

```
>>> def gen_squares(max_root):
...     for x in range(max_root):
...         yield x**2
...
>>> for square in gen_squares(5):
...     print(square)
...
0
1
4
9
16
```

Generator Functions & Objects

The function with `yield` is called a **generator function**.

The object it returns is called a **generator object**.

```
>>> def gen_squares(max_root):
...     for x in range(max_root):
...         yield x**2
...
>>> squares = gen_squares(5)
>>> type(squares)
<class 'generator'>
```

Pop quiz

Create a new file called `gensquares.py`. Type this in and run it:

```
def gen_squares(max_root):
    for x in range(max_root):
        yield x**2
squares = gen_squares(5)
for square in squares: print(square)
```

It should print:

```
0
1
4
9
16
```

When done, give a thumbs up, comment out the `for` loop, and replace it with `print(next(squares))` repeated several times. What does that do?

The next() thing

```
>>> squares = gen_squares(5)
>>> next(squares)
0
>>> next(squares)
1
>>> next(squares)
4
>>> next(squares)
9
>>> next(squares)
16
>>> next(squares)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Future-proofing "next"

```
def gen_up_to(limit):
    n = 0
    while n <= limit:
        yield n
        n += 1

it = gen_up_to(10)

# Works in Python 3 only
it.__next__()
# Works in Python 2 only
it.next()
# Works in Python 2, 3, 4, ...
next(it)
# next() also lets you supply a default value
next(it, None)
```

Multiple Yields

You can have more than one yield statement.

```
>>> def myitems(top):
...     while top > 0:
...         yield top**2
...         top -= 1
...     yield "All done"
...
>>> for item in myitems(3):
...     print(item)
...
9
4
1
All done
```

Lab: Generators

Lab file: `generators/generators.py`

- In labs/py3 for 3.x; labs/py2 for 2.7
- When you are done, give a thumbs up...
- ... and then do `generators/generators_extra.py`

NOTE: If the test fails saying it sees `<class 'generator'>`, but expected `<type 'generator'>` - or the other way around - check your Python version.

Scalable Generators

Here's another way to implement `myitems`:

```
>>> def myitems(top):
...     for x in range(top, 0, -1):
...         yield x**2
...     yield "All done"
...
>>> for item in myitems(3):
...     print(item)
...
9
4
1
All done
```

Same output. But ... is there a problem hiding here?

Iterator Protocol

Any object in Python can be an iterator. It just needs to define proper `__iter__` and `__next__` methods.

```
class Squares:
    def __init__(self, max_root):
        self.max_root = max_root
        self.root = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.root == self.max_root:
            raise StopIteration
        value = self.root ** 2
        self.root += 1
        return value

for square in Squares(5):
    print(square)
```

We call this the *iterator protocol*.

Generator Patterns

Generator Funcs and Files

Here's a little program. Focus on `matchinglines()`:

```
# findpattern.py
import sys

def matchinglines(pattern, path):
    with open(path) as handle:
        for line in handle:
            if pattern in line:
                yield line.rstrip('\n')

pattern, path = sys.argv[1], sys.argv[2]
for line in matchinglines(pattern, path):
    print(line)
```

Using matchinglines

`log.txt`:

```
WARNING: Disk usage exceeding 85%
DEBUG: User 'tinytim' upgraded to Pro version
INFO: Sent email campaign, completed normally
WARNING: Almost out of beer
```

`matchinglines("WARNING:", "log.txt")` yields:

```
WARNING: Disk usage exceeding 85%
WARNING: Almost out of beer
```

It does this on ANY size log file, with a small memory footprint.

Records

The log file contains **records**, one per line:

```
WARNING: Disk usage exceeding 85%
DEBUG: User 'tinytim' upgraded to Pro version
...
```

Suppose your application needs that data in dict form:

```
{"level": "WARNING", "message": "Disk usage exceeding 85%"}
{"level": "DEBUG", "message": "User 'tinytim' upgraded to Pro version"}
```

How do you scalably transform log lines to dictionaries?

Transformation

You want to **transform** the data, from one form to another:

```
def parse_log_records(lines):
    for line in lines:
        level, message = line.split(": ", 1)
        yield {"level": level, "message": message}
```

Chain it together with `matchinglines`:

```
log_lines = matchinglines("WARNING:", "logfile.txt")
for record in parse_log_records(log_lines):
    print(record)
```

```
{'level': 'WARNING', 'message': 'Disk usage exceeding 85%'}
{'level': 'WARNING', 'message': 'Almost out of beer'}
```

Building Blocks

But it can also be used on its own.

```
with open("logfile.txt") as handle:
    for record in parse_log_records(handle):
        print(record)
```

```
{'level': 'WARNING', 'message': 'Disk usage exceeding 85%'}
{'level': 'DEBUG', 'message': 'User "tinytim" upgraded to Pro version'}
{'level': 'INFO', 'message': 'Sent email campaign, completed normally'}
{'level': 'WARNING', 'message': 'Almost out of beer'}
```

`matchinglines` and `parse_log_records` are building blocks, which can be used to build different data processing streams.

Scalable Composability

Call this *scalable composability*.

It goes beyond designing composable functions and types.

Ask yourself how you can make the components scalable, **and** whatever is assembled out of them scalable too.

Generator Functions!

Interfaces

How you might integrate these in a class:

```
class Logs:
    def __init__(self, logfile_path):
        self.logfile_path = logfile_path
    def records(self):
        with open(self.logfile_path) as log_lines:
            for record in parse_log_records(log_lines):
                yield record
def warnings(self):
    log_lines = matchinglines("WARNING:", self.logfile_path)
    for record in parse_log_records(log_lines):
        yield record
```

Transforming Adapter

You can think of `parse_log_records` as an **adapter**... transforming records in one form (lines) to a more useful one (dictionaries).

Well-structured programs have many such boundaries of transformation.

Generator functions are an excellent device for creating them.

Comprehensions (our next topic) are surprisingly relevant.

Record Mapping

You can think of generator functions as mapping one stream of records to another stream.

With `parse_log_records`, one input record maps to one output record:

- **Input record**: one *line*
- **Output record**: one *dict*

What happens when the mapping **isn't** one-to-one?

- Several input records are consumed to produce one output record. Or...
- One input record creates several output records

Fan Out: Word Parsing

Imagine a text file containing lines in a poem:

```
all night our room was outer-walled with rain
drops fell and flattened on the tin roof
and rang like little disks of metal
...
```

Let's create a generator function, `words_in_text`, producing the words one at a time.

words_in_text

First approach:

```
def words_in_text(path):
    with open(path) as handle:
        for line in handle:
            line = line.rstrip('\n')
            for word in line.split():
                yield word
```

There is a potential bottleneck in here. What is it? How can we do better?

Looooong lines

```
def words_in_text(path):
    BUFFER_SIZE = 2**20
    def read(): return handle.read(BUFFER_SIZE)
    def normalize(chunk): return chunk.lower().rstrip(',!.\n')
    with open(path) as handle:
        buffer = read()
        start, end = 0, -1
        while True:
            for match in re.finditer(r'[ \t\n]+', buffer):
                end = match.start()
                yield normalize(buffer[start:end])
                start = match.end()
            new_buffer = read()
            if new_buffer == '':
                break # end of file
            buffer = buffer[end+1:] + new_buffer
            start, end = 0, -1
word = normalize(buffer[start:])
if word != '':
    yield word
```

Fan In: House Sale Data

`housedata.txt`: One key-value pair per line, with records separated by blank lines.

```
address: 1423 99th Ave
square_feet: 1705
price_usd: 340210

address: 24257 Pueblo Dr
square_feet: 2305
price_usd: 170210

address: 127 Cochran
square_feet: 2068
price_usd: 320500
```

Generating House Records

We want a generator function called `house_records`, which will read this data in, and give us a stream of dictionaries:

```
>>> houses = house_records("housedata.txt")
>>> house = next(houses)
>>> type(house)
<class 'dict'>
>>> list(house.keys())
['address', 'square_feet', 'price_usd']
>>> house['address']
'1423 99th Ave'
>>> house = next(houses)
>>> house['address']
'24257 Pueblo Dr'
```

Reading House Records

```
def house_records(path):
    with open(path) as lines:
        record = {}
        for line in lines:
            if line == '\n':
                yield record
                record = {}
                continue
            key, value = line.rstrip('\n').split(': ', 1)
            record[key] = value
        yield record
```

Lab: Generator Adapters

Lab file: `generators/adapter.py`

- In labs/py3 for 3.x; labs/py2 for 2.7
- When you are done, give a thumbs up...
- ... and then do `generators/adapter_extra.py`

The methods of str are detailed here:
https://docs.python.org/3/library/stdtypes.html#string-methods

List Comprehensions (And Other Comprehensions)

What is a list comprehension?

A **list comprehension** is a way to create a list in Python.

It **high level** and **declarative**.

Squares

```
>>> squares = []
>>> for x in range(5):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16]
```

Higher Level

As a **list comprehension**:

```
>>> [ x**2 for x in range(5) ]
[0, 1, 4, 9, 16]
```

Exactly equivalent to this:

```
>>> squares = []
>>> for x in range(5):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16]
```

Structure

```
[ EXPR for VAR in SEQ ]
```

```
>>> [ x**2 for x in range(5) ]
[0, 1, 4, 9, 16]
```

- Expression (in terms of variable)
- The name of that variable
- The source sequence

EXPR and SEQ

```
[ EXPR for VAR in SEQ ]
```

EXPR can be any Python expression:	SEQ can be:
• Arithmetic expressions like `n+3` • A function call like `f(m)`, using `m` as the variable • A slice operation (like `s[::-1]`, to reverse a string) • Method calls (`foo.bar()`, iterating over a sequence of objects)	• A list or tuple • A generator object • Any iterator • Even another comprehension

Examples

```
>>> [ 2*m+3 for m in range(10, 20, 2) ]
[23, 27, 31, 35, 39]

>>> numbers = [ 9, -1, -4, 20, 11, -3 ]
>>> [ abs(num) for num in numbers ]
[9, 1, 4, 20, 11, 3]

>>> pets = ["dog", "parakeet", "cat", "llama"]
>>> [ pet.upper() for pet in pets ]
['DOG', 'PARAKEET', 'CAT', 'LLAMA']

>>> def repeat(s):
...     return s + s
>>> [ repeat(pet) for pet in pets ]
['dogdog', 'parakeetparakeet', 'catcat', 'llamallama']
```

Practice the syntax

Type in the following on a Python prompt:

```
>>> colors = ["red", "green", "blue"]
>>> [ z*3 for z in range(5) ]
[0, 3, 6, 9, 12]

>>> [ abs(x) for x in range(-3, 3) ]
[3, 2, 1, 0, 1, 2]

>>> [ color.upper() for color in colors ]
['RED', 'GREEN', 'BLUE']
```

Multiple for's

Comprehensions can have several `for` clauses.

```
>>> colors = ["orange", "purple", "pink"]
>>> toys = ["bike", "basketball", "skateboard", "doll"]
>>>
>>> [ color + " " + toy
...   for color in colors
...   for toy in toys ]
['orange bike', 'orange basketball', 'orange skateboard', 'orange doll',
'purple bike', 'purple basketball', 'purple skateboard', 'purple doll', 'pink
bike', 'pink basketball', 'pink skateboard', 'pink doll']
```

Chaining "for" clauses

You can chain multiple `for` clauses together, effectively generating the source sequence.

```
>>> ranges = [range(1, 7), range(4, 12, 3), range(-5, 9, 4)]
>>> [ float(num)
...   for subrange in ranges
...   for num in subrange ]
[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 4.0, 7.0, 10.0, -5.0, -1.0, 3.0, 7.0]
```

But order matters.

```
>>> [ float(num)
...   for num in subrange
...   for subrange in ranges ]
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: name 'subrange' is not defined
```

Chaining Vs. Combos

If the variables in the `for` clauses overlap, you're chaining. The expression depends only on the variable in the last clause.

```
[ float(num)
  for subrange in ranges
  for num in subrange ]
```

If no overlap, the expression will depend on *all* the for-clause variables. Mathematically, it's like an outer product.

```
[ color + " " + toy
  for color in colors
  for toy in toys ]
```

For this last one, does order matter?

Ordering "for" clauses

```
>>> colors = ["orange", "purple", "pink"]
>>> toys = ["bike", "basketball", "skateboard", "doll"]
>>>
>>> [ color + " " + toy
...   for color in colors
...   for toy in toys ]
['orange bike', 'orange basketball', 'orange skateboard', 'orange doll',
'purple bike', 'purple basketball', 'purple skateboard', 'purple doll', 'pink
bike', 'pink basketball', 'pink skateboard', 'pink doll']
>>>
>>> [ color + " " + toy
...   for toy in toys
...   for color in colors ]
['orange bike', 'purple bike', 'pink bike', 'orange basketball', 'purple
basketball', 'pink basketball', 'orange skateboard', 'purple skateboard',
'pink skateboard', 'orange doll', 'purple doll', 'pink doll']
```

`for` clause order affects the resulting element order. You can choose the order if the `for` clauses are independent (not chained).

Filtering

List comprehensions can exclude elements.

```
>>> numbers = [ 9, -1, -4, 20, 11, -3 ]
>>>
>>> # Positive numbers:
... [ x for x in numbers if x > 0 ]
[9, 20, 11]
>>>
>>> # Squares of even numbers:
... [ x**2 for x in numbers if x % 2 == 0 ]
[16, 400]
```

"If" Structure

```
[ EXPR for VAR in SEQ if CONDITION ]
```

```
>>> [ x+1 for x in numbers if x > 0 ]
[10, 21, 12]
>>> [ x**2 for x in numbers if x % 2 == 0 ]
[16, 400]
```

More complex If

```
>>> def is_palindrome(s):
...     return s == s[::-1]
>>> words = ["bias", "dad", "eye", "deed", "tooth"]
>>> palindromes = [ word for word in words
...                 if is_palindrome(word)]
>>> print(palindromes)
['dad', 'eye', 'deed']
```

Function of VAR

```
[ EXPR for VAR in SEQ if CONDITION ]
```

In general, both EXR and CONDITION will be a function of VAR.

```
[ some_expr(x) for x in some_seq
  if some_condition(x)]
```

One of the only exceptions:

```
# List of ten 0's
[ 0 for x in range(10) ]
```

Required Syntax

You must have the **for** and **in** keywords, always. (And **if** if you're filtering). Even if the expression and variable are the same.

```
>>> # Like this:
... [ x for x in numbers if x > 3 ]
[9, 20, 11]
>>> # Nope:
... [ x in numbers if x > 3 ]
  File "<stdin>", line 2
    [ x in numbers if x > 3 ]
                   ^
SyntaxError: invalid syntax
```

Practice the syntax

```
>>> numbers = [ 9, -1, -4, 20, 11, -3 ]
>>> def is_even(n):
...     return n % 2 == 0
...
>>> [ x for x in numbers if x > 0 ]
[9, 20, 11]
>>>
>>> [ x*2 for x in numbers if x < 10 ]
[18, -2, -8, -6]
>>>
>>> [ 20-x for x in numbers if is_even(x)]
[24, 0]
```

Benefits

- Very readable
- Low cognitive overhead
- Very maintainable

Indentation

You can (and should!) split the comprehension across multiple lines.

```
def double_short_words(words):
    return [ word + word
             for word in words
             if len(word) < 5 ]
```

Indendation 2

Python's normal whitespace rules are suspended within a list comprehension's brackets. Here's another way to do it:

```
def double_short_words(words):
    return [
        word + word
        for word in words
        if len(word) < 5
    ]
```

Multiple for's and if's

```
>>> weights = [0.2, 0.5, 0.9]
>>> values = [27.5, 13.4]
>>> offsets = [4.3, 7.1, 9.5]
>>>
>>> [ (weight, value, offset)
...   for weight in weights
...   for value in values
...   for offset in offsets ]
[(0.2, 27.5, 4.3), (0.2, 27.5, 7.1), (0.2, 27.5, 9.5), (0.2, 13.4, 4.3),
(0.2, 13.4, 7.1), (0.2, 13.4, 9.5), (0.5, 27.5, 4.3), (0.5, 27.5, 7.1), (0.5,
27.5, 9.5), (0.5, 13.4, 4.3), (0.5, 13.4, 7.1), (0.5, 13.4, 9.5), (0.9, 27.5,
4.3), (0.9, 27.5, 7.1), (0.9, 27.5, 9.5), (0.9, 13.4, 4.3), (0.9, 13.4, 7.1),
(0.9, 13.4, 9.5)]
>>> [ (weight, value, offset)
...   for weight in weights
...   for value in values
...   for offset in offsets
...   if offset > 5.0
...   if weight * value < offset ]
[(0.2, 27.5, 7.1), (0.2, 27.5, 9.5), (0.2, 13.4, 7.1), (0.2, 13.4, 9.5),
(0.5, 13.4, 7.1), (0.5, 13.4, 9.5)]
```

Lab: List Comprehensions

Lab file: `comprehensions/listcomp.py`

- In labs/py3 for 3.x; labs/py2 for 2.7
- When you are done, give a thumbs up...
- ... then do `comprehensions/listcomp_extra.py`

Other Comprehensions

Dictionary comprehensions:

```
>>> blocks = { num: "x" * num for num in range(5) }
>>> print(blocks)
{0: '', 1: 'x', 2: 'xx', 3: 'xxx', 4: 'xxxx'}
```

Set comprehensions

Imagine a Student class, which includes a "major" attribute.

```
>>> # A list of student majors...
... [ student.major for student in students ]
['Computer Science', 'Economics', 'Computer Science', 'Economics', 'Basket
Weaving']
>>> # And a set of majors:
... { student.major for student in students }
{'Economics', 'Computer Science', 'Basket Weaving'}
>>> # You can also use the set() built-in.
... set(student.major for student in students)
{'Economics', 'Computer Science', 'Basket Weaving'}
```

Generator Comprehensions

If you use parentheses instead of square brackets, something really interesting happens.

```
>>> items = ( x*3 for x in range(5) )
>>> type(items)
<class 'generator'>
>>> next(items)
0
>>> next(items)
3
>>> next(items)
6
```

Watch Out: The official docs call these "generator expressions". But a lot of Pythonistas call them "generator comprehensions". Because they ARE comprehensions.

Generator Comprehensions

```
>>> list_comp = [ x*3 for x in range(5) ]
>>> type(list_comp)
<class 'list'>
>>> gen_expr = ( x*3 for x in range(5) )
>>> type(gen_expr)
<class 'generator'>
```

Another way to generate

It turns out that this...

```
COUNT = 5
items = ( x*3 for x in range(COUNT) )
```

... is EXACTLY equivalent to this:

```
def gen_items(limit):
    for x in range(limit):
        yield x*3

COUNT = 5
items = gen_items(COUNT)
```

((Pro Tip))

When passing a generator comprehension inline to a function, you can omit the parenthesis:

```
>>> sorted( (student.name for student in students) )
['Jones, Tina', 'Shan, Geetha', 'Simmons, Russell', 'Smith, Joe']
>>> sorted( student.name for student in students )
['Jones, Tina', 'Shan, Geetha', 'Simmons, Russell', 'Smith, Joe']
```

But not always:

```
>>> sorted( student.name for student in students, reversed=True)
  File "<stdin>", line 1
SyntaxError: Generator expression must be parenthesized if not sole argument
```

Rule of thumb: `( ( ... ) )` can be replaced with `( ... )`

Objects in Python

Intro to OOP

This is a quick yet in-depth review of Python's syntax for object-oriented programming.

For those more experienced: we'll include some valuable, but rarely-known secrets along the way.

Creating classes

Use the `class` keyword. `__init__` is the constructor.

All methods, including `__init__`, take `self` as their first argument. Similar to "`this`" in other languages, but explicit.

```
# Python 3
class Pet:
    def __init__(self, name):
        self.name = name
```

Create objects by calling the class name like a function - no "new" keyword. The "self" parameter is automatically inserted.

```
>>> pet = Pet("Fido")
>>> pet.name
'Fido'
```

Inheritance

Declare subclasses on the `class` line, with parentheses.

```
# Python 3, still
class Pet:
    def __init__(self, name):
        self.name = name
class Dog(Pet):
    pass
class Cat(Pet):
    pass
```

```
>>> pet = Dog("Fido")
>>> pet.name
'Fido'
>>> isinstance(pet, Pet)
True
>>> isinstance(pet, Dog)
True
>>> isinstance(pet, Cat)
False
```

`isinstance` is a built-in function you can use to check object types

Methods

Methods are defined just like functions, except:

- Indented in the class, and
- take `self` as the first argument.

`self` is always implicitly prepended.

```
class Person:
    def __init__(self, first, last):
        self.first = first
        self.last = last
    def full_name(self):
        return self.first + " " + self.last
    def formal_name(self, title):
        return title + " " + self.full_name()
```

```
>>> guy = Person("John", "Smith")
>>> guy.formal_name("Mr.")
'Mr. John Smith'
```

Everything inherits from "object"

In modern Python, all classes need to inherit from a built-in base class called `object`. In Python 2, you must explicity declare it:

```
>>> # Python 2
... class PetFromObject(object):
...     def __init__(self, name):
...         self.name = name
>>> class PetNotFromObject:
...     def __init__(self, name):
...         self.name = name
>>> issubclass(PetFromObject, object)
True
>>> issubclass(PetNotFromObject, object)
False
```

Everything inherits from "object"

In Python 3, all classes automatically inherit from `object`.

```
>>> # Python 3
... class PetFromObject:
...     def __init__(self, name):
...         self.name = name
>>> issubclass(PetFromObject, object)
True
```

Inheriting from `object` gives you many benefits, and no downsides (unless you are working with ancient legacy Python code).

One benefit: your subclass methods can invoke the superclass version.

Superclass methods

In Python 3, invoke the superclass' method using `super()`.

```
class LapDog:
    def speak(self):
        return "Yip!"
class LoudLapDog(LapDog):
    def speak(self):
        sound = super().speak().upper()
        return sound + sound + sound
```

```
>>> fifi = LoudLapDog()
>>> fifi.speak()
'YIP!YIP!YIP!'
```

Superclass methods (Py 2)

In Python 2, you need to pass arguments to `super()`.

```
class LapDog(object): # Note the (object)
    def speak(self):
        return "Yip!"
class LoudLapDog(LapDog):
    def speak(self):
        # super(LoudLapDog, self).speak refers to LapDog.speak
        sound = super(LoudLapDog, self).speak().upper()
        return sound + sound + sound
```

```
>>> fifi = LoudLapDog()
>>> fifi.speak()
'YIP!YIP!YIP!'
```

Default Variables

You can define member variables in the class directly. Each instance gets its own copy to modify.

```
>>> class Coin:
...     value = 1
...
>>> penny = Coin()
>>> penny.value
1
>>>
>>> nickel = Coin()
>>> nickel.value = 5
>>> nickel.value
5
>>>
>>> another_penny = Coin()
>>> another_penny.value
1
```

Overriding Values

A subclass can change the value.

```
class Pet:
    sound = ""
    def speak(self):
        print("The pet says: " + self.sound)
class Dog(Pet):
    sound = "Woof!"
class Cat(Pet):
    sound = "Meow"
class Turtle(Pet):
    pass
```

```
>>> Dog().speak()
'The pet says: Woof!'
>>> Cat().speak()
'The pet says: Meow'
>>> Turtle().speak()
'The pet says: '
```

No Multiple Dispatch

In some languages, you can define a method twice with different signatures.

Not in Python. The second definition **silently masks** the first one.

```
>>> class BarkingDog:
...     def speak(self):
...         return "Bark"
...     def speak(self, punctuation):
...         return "BARK" + punctuation
...
>>> jojo = BarkingDog()
>>> jojo.speak("!")
'BARK!'
>>> jojo.speak()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: speak() missing 1 required positional argument: 'punctuation'
```

"Protected" Attributes

Python doesn't have access control modifiers, like Java, C# or C++.

Convention: prefixing with a single underscore means "don't rely on this being available."

```
class SignalParser:
    def __init__(self):
        self._state = 'waiting'
    def receive(self):
        self._state = 'receiving'
```

But the language doesn't enforce it.

```
>>> sp = SignalParser()
>>> sp.receive()
>>> print("Mwahaha, I can see your hidden state is " + sp._state)
Mwahaha, I can see your hidden state is receiving
```

A step better

You can prefix the member attribute or method with __, and Python will make it harder to access.

```
class SignalParser:
    def __init__(self):
        self.__state = 'waiting'
    def receive(self):
        self.__state = 'receiving'
```

```
>>> sp = SignalParser()
>>> sp.receive()
>>> print("I cannot read your hidden state! " + sp.__state)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'SignalParser' object has no attribute '__state'
```

But actually, it just mangles the name:

```
>>> print("Oh wait, I CAN! " + sp._SignalParser__state)
Oh wait, I CAN! receiving
```

Special Methods

Python's object system has special methods you can implement, surrounded by pairs of underscores. These are called *magic methods*.

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
    def __str__(self):
        return "${}.{:02}".format(self.dollars, self.cents)
```

```
>>> print(Money(2,3))
$2.03
```

In speech, we say "dunder str", rather than "underscore underscore str underscore underscore".

Properties In Python

Properties

A hybrid between a method and member variable.

```
class Person:
    def __init__(self, first, last):
        self.first = first
        self.last = last
    @property
    def full_name(self):
        return self.first + " " + self.last
```

Dynamic Attribute

Even though it's defined as a method, you access it like a member variable.

```
>>> guy = Person("Joe", "Smith")

>>> guy.full_name
'Joe Smith'

>>> guy.full_name()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object is not callable
```

In fact, you **can't** call it like a method, even if you want to.

Read-Only

By default, a property is read-only.

```
>>> guy.full_name
'Joe Smith'
>>> guy.full_name = 'John Doe'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

This is either a feature or a bug, depending on what you want.

Setters

You can make a property writable with a **setter** - an extra, specially-marked method.

```
class Person:
    def __init__(self, first, last):
        self.first = first
        self.last = last

    @property
    def full_name(self):
        return self.first + " " + self.last

    @full_name.setter
    def full_name(self, value):
        first, last = value.split(" ")
        self.first = first
        self.last = last
```

Setting Properties

```
    @full_name.setter
    def full_name(self, value):
        first, last = value.split(" ")
        self.first = first
        self.last = last
```

```
>>> guy = Person("Joe", "Smith")
>>> guy.full_name = "Sam Jones"
>>> print(guy.first)
Sam
>>> print(guy.last)
Jones
>>> print(guy.full_name)
Sam Jones
```

Practice: getset.py

```
class Person: # or "Person(object):" for Python 2
    def __init__(self, first, last):
        self.first = first
        self.last = last
    @property
    def full_name(self):
        return self.first + " " + self.last
    @full_name.setter
    def full_name(self, value):
        first, last = value.split(" ")
        self.first = first
        self.last = last
guy = Person("Joe", "Smith")
print(guy.full_name)
guy.full_name = "Sam Jones"
print(guy.last + ", " + guy.first)
```

```
# Running "python3 getset.py" should have this output:
Joe Smith
Jones, Sam
```

Read-Only Pattern

A common Python design pattern: Use `@property` to create a read-only attribute.

```
class Ticket:
    def __init__(self, price):
        self._price = price
    @property
    def price(self):
        return self._price
```

```
>>> ticket = Ticket(42)
>>> ticket.price
42
>>> ticket.price = 41
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

Validation

Useful setter pattern: ensure a value is set to the correct range.

```
class Ticket:
    def __init__(self, price):
        self._price = price
    @property
    def price(self):
        return self._price
    @price.setter
    def price(self, new_price):
        # Only allow positive prices.
        if new_price < 0:
            raise ValueError("Nice try")
        self._price = new_price
```

Validation

This will raise a run-time error if we try to cheat:

```
# In class Ticket...
    @price.setter
    def price(self, new_price):
        # Only allow positive prices.
        if new_price < 0:
            raise ValueError("Nice try")
        self._price = new_price
```

```
>>> t = Ticket(42)
>>> t.price
42
>>> t.price = -1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 12, in price
ValueError: Nice try
```

Validation

Pop quiz: Do you see a way to improve the constructor?

What's the potential problem lurking in this code?

```
class Ticket:
    def __init__(self, price):
        self._price = price
    @property
    def price(self):
        return self._price
    @price.setter
    def price(self, new_price):
        # Only allow positive prices.
        if new_price < 0:
            raise ValueError("Nice try")
        self._price = new_price
```

Use Setters In Your Methods

You can use an object's setters in your own methods.

For `Ticket`, this lets us get the validation check at object-creation time!

```
class Ticket:
    def __init__(self, price):
        # instead of "self._price = price"
        self.price = price
    @property
    def price(self):
        return self._price
    @price.setter
    def price(self, new_price):
        # Only allow positive prices.
        if new_price < 0:
            raise ValueError("Nice try")
        self._price = new_price
```

Lab: Properties

Lab file: `patterns/properties.py`

- In labs/py3 for 3.x; labs/py2 for 2.7
- When you are done, give a thumbs up...
- ... and then do `patterns/properties_extra.py`

Properties And Refactoring

Here's a money class.

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
    # And some other methods...
```

Imagine this is released as a library, and many teams are using and relying on this class.

Refactor

One day, we decide to internally just keep track of cents.

```
class Money:
    def __init__(self, dollars, cents):
        self.total_cents = dollars * 100 + cents
```

That creates a maintainability problem. Can you spot it?

The problem

Other code referencing its attributes suddenly breaks.

```
money = Money(27, 12)
template = "I have {:d} dollars and {:d} cents."
# This line breaks, because there's no longer
# dollars or cents attributes.
print(template.format(money.dollars, money.cents))
```

This could be a major impediment to changing the class interface.

The solution

But not in Python.

```
class Money:
    def __init__(self, dollars, cents):
        self.total_cents = dollars * 100 + cents
    # Getter and setter for dollars...
    @property
    def dollars(self):
        return self.total_cents // 100
    @dollars.setter
    def dollars(self, new_dollars):
        self.total_cents =
            100 * new_dollars + self.cents
```

Properties for Refactoring

```
    # And the getter and setter for cents.
    @property
    def cents(self):
        return self.total_cents % 100
    @cents.setter
    def cents(self, new_cents):
        self.total_cents =
            100 * self.dollars + new_cents
```

Design Patterns in Python

Python Patterns

Object-oriented design patterns work differently in Python than other languages, because of Python's very different feature set.

Observer pattern

Defines a "one to many" relationship among objects.

- One central object, called **the observable**, watches for events.
- Another set of objects, the **observers**, ask the observable to tell them when that event happens.

PubSub

There's another name for this: "Pub-Sub".

- One central object, called **the publisher**, watches for events.
- Another set of objects, the **subscribers**, ask the publisher to tell them when that event happens.

To me, that's a better name. So in working with the observer pattern, we'll speak of "publishers" and "subscribers".

Let's start with the simple observer pattern.

Subscriber

In the simplest form, each subscriber has a method named `update`, which takes a message.

```
class Subscriber:
    def __init__(self, name):
        self.name = name
    def update(self, message):
        print('{} got message "{}"'.format(self.name, message))
```

The publisher invokes that update method.

Registration

The subscriber must tell the publisher it wants to get messages. So the publisher object has a `register` method.

```
class Publisher:
    def __init__(self):
        self.subscribers = set()
    def register(self, who):
        self.subscribers.add(who)
    def unregister(self, who):
        self.subscribers.discard(who)
```

Sending Messages

When an event happens, you have the publisher send the message to all subscribers using a `dispatch` method.

```
class Publisher:
    def __init__(self):
        self.subscribers = set()
    def register(self, who):
        self.subscribers.add(who)
    def unregister(self, who):
        self.subscribers.discard(who)
    def dispatch(self, message):
        for subscriber in self.subscribers:
            subscriber.update(message)
```

Using in Code

```
pub = Publisher()

bob = Subscriber('Bob')
alice = Subscriber('Alice')
john = Subscriber('John')

pub.register(bob)
pub.register(alice)
pub.register(john)

pub.dispatch("It's lunchtime!")
pub.unregister(john)
pub.dispatch("Time for dinner")
```

Output

```
# from last slide:
pub.dispatch("It's lunchtime!")
pub.unregister(john)
pub.dispatch("Time for dinner")
```

```
John got message "It's lunchtime!"
Bob got message "It's lunchtime!"
Alice got message "It's lunchtime!"
Bob got message "Time for dinner"
Alice got message "Time for dinner"
```

Other forms

This is the simplest form of the observer pattern in Python.

Advantage: Very little code. Easy to set up.

Disadvantage: Inflexible. Subscribers must be of classes implementing an `update` method.

Also: simplistic. Publisher notifies on just one kind of event.

If we go more complex, what does that buy us?

Alt Callback

In Python, *everything* is an object. Even methods.

So subscriber can register a method other than `update`.

```
# This subscriber uses the standard "update"
class SubscriberOne:
    def __init__(self, name):
        self.name = name
    def update(self, message):
        print('{} got message "{}"'.format(self.name, message))
# This one wants to use "receive"
class SubscriberTwo:
    def __init__(self, name):
        self.name = name
    def receive(self, message):
        print('{} got message "{}"'.format(self.name, message))
```

Alt Callback: Publisher

```
class Publisher:
    def __init__(self):
        self.subscribers = dict()
    def register(self, who, callback=None):
        if callback is None:
            callback = who.update
        self.subscribers[who] = callback
    def dispatch(self, message):
        for callback in self.subscribers.values():
            callback(message)
    def unregister(self, who):
        del self.subscribers[who]
```

Using

```
pub = Publisher()
bob = SubscriberOne('Bob')
alice = SubscriberTwo('Alice')
john = SubscriberOne('John')

pub.register(bob)
pub.register(alice, alice.receive)
pub.register(john, john.update)

pub.dispatch("It's lunchtime!")
pub.unregister(john)
pub.dispatch("Time for dinner")
```

Output

```
# from last slide:
pub.dispatch("It's lunchtime!")
pub.unregister(john)
pub.dispatch("Time for dinner")
```

```
Alice got message "It's lunchtime!"
John got message "It's lunchtime!"
Bob got message "It's lunchtime!"
Alice got message "Time for dinner"
Bob got message "Time for dinner"
```

Channels

The publishers so far only do "all or nothing" notification.

What about one publisher that can watch several event types? How could we implement this?

For this, let's use the regular "update" subscriber:

```
class Subscriber:
    def __init__(self, name):
        self.name = name
    def update(self, message):
        print('{} got message "{}"'.format(self.name, message))
```

Publisher: channels

```
class Publisher:
    def __init__(self, channels):
        # Create an empty subscribers dict
        # for every channel
        self.channels = { channel : dict()
                          for channel in channels }
    def register(self, channel, who, callback=None):
        if callback is None:
            callback = who.update
        subscribers = self.channels[channel]
        subscribers[who] = callback
```

Publisher: channels

```
    def dispatch(self, channel, message):
        subscribers = self.channels[channel]
        for callback in subscribers.values():
            callback(message)
```

Publisher: channels

```
pub = Publisher(['lunch', 'dinner'])
bob = Subscriber('Bob')
alice = Subscriber('Alice')
john = Subscriber('John')

pub.register("lunch", bob)
pub.register("dinner", alice)
pub.register("lunch", john)
pub.register("dinner", john)

pub.dispatch("lunch", "It's lunchtime!")
pub.dispatch("dinner", "Dinner is served")
```

Publisher: channels

```
# from last slide:
pub.dispatch("lunch", "It's lunchtime!")
pub.dispatch("dinner", "Dinner is served")
```

```
Bob got message "It's lunchtime!"
John got message "It's lunchtime!"
Alice got message "Dinner is served"
John got message "Dinner is served"
```

Labs: Patterns

Let's do a more self-directed lab. You're going to use the observer pattern to implement a program called `filewatch.py`.

Instructions: `patterns/filewatch-lab.txt`

- In labs/py3 for 3.x; labs/py2 for 2.7
- First follow the instructions to write `filewatch.py`
- When you are done, give a thumbs up...
- ... and then follow the further instructions for `filewatch_extra.py`

The idea of factories

Imagine this version of a money class:

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
```

This constructor expects both dollar and cent amounts.

Constructor mismatch

What if our application is working with cents directly? We have to manually decompose it:

```
>>> # Emptying the penny jar...
... total_pennies = 3274
>>> # // is integer division
... dollars = total_pennies // 100
>>> cents = total_pennies % 100
>>> total_cash = Money(dollars, cents)
```

Suppose this is very common in our code. Can we encapsulate it a bit better?

Change the constructor?

One thing we can do is change the constructor:

```
class Money:
    def __init__(self, total_cents):
        self.dollars = total_cents // 100
        self.cents = total_cents % 100
```

That means we lose the first constructor, though.

(Some languages let you define several constructors. But even if Python let us do that, it would not solve all problems.)

Factory function

A better solution: keep the more general constructor, and create a "factory" function.

```
# Let's back up, to the original Money constructor.
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents

# From cents:
def money_from_pennies(total_cents):
    dollars = total_cents // 100
    cents = total_cents % 100
    return Money(dollars, cents)
```

As many as we want!

In fact, we can create as many of these factory functions as we want. For example, create `Money` from a string like "$140.75":

```
import re
def money_from_string(amount):
    # amount is a string like "$140.75"
    match = re.search(r'^\$(?P<dollars>\d+)\.(?P<cents>\d\d)$', amount)
    if match is None:
        raise ValueError('Invalid amount: {}'.format(amount))
    dollars = int(match.group('dollars'))
    cents = int(match.group('cents'))
    return Money(dollars, cents)
```

This works. But...

Subclassing

... it only works for the `Money` class. Subclasses need a whole different set of functions.

(And if we change the class name to, say, `Dollars`, we have a bit more refactoring to do too.)

Python provides a better solution.

@classmethod

`classmethod` is a built-in decorator that is applied to class methods. The method becomes associated with the class itself.

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
    @classmethod
    def from_pennies(cls, total_cents):
        dollars = total_cents // 100
        cents = total_cents % 100
        return cls(dollars, cents)
```

Notice the first argument of `from_pennies`.

Class methods

You call it off the *class itself*, not an instance of the class.

```
>>> # It's like an extra constructor.
... piggie_bank_cash = Money.from_pennies(3217)
>>> type(piggie_bank_cash)
<class '__main__.Money'>
>>> piggie_bank_cash.dollars
32
>>> piggie_bank_cash.cents
17
>>> # And we can define as many as we want.
... piggie_bank_cash = Money.from_string("$14.72")
```

Subclassing

This automatically works with subclasses:

```
>>> class TipMoney(Money):
...     pass
...
>>> tip = TipMoney.from_pennies(475)
>>> type(tip)
<class '__main__.TipMoney'>
```

More maintainable. `@classmethod` is worth keeping in your toolbox.

Advantages

The OOP literature calls this the "simple factory" pattern. I prefer to call it "alternate constructor".

Its advantages:

- Can use descriptive method names
- Automatically extends to subclasses
- Encapsulated in the pertinent class

Other factories

- "factory method" pattern (dynamic type pattern)
- "abstract factory" pattern (more complex, can be useful for DI)

Static Methods

You can use static methods in Python too.

They tend to be less useful in Python than in other languages, because of `@classmethod` and other reasons.

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents

    @classmethod
    def from_pennies(cls, total_cents):
        dollars, cents = cls.pennies2dollarsandcents(total_cents)
        return cls(dollars, cents)

    # Utility function. Could also be a standalone function
    # in the same module as this class.
    @staticmethod
    def pennies2dollarsandcents(pennies):
        dollars = pennies // 100
        cents = pennies % 100
        return (dollars, cents)
```

Dictionary Views & Iteration

Dictionary Views & Iteration

Here's a Python 3 dictionary:

```
>>> calories = {
...     "apple": 95,
...     "slice of bacon": 43,
...     "cheddar cheese": 113,
...     "ice cream": 15, # You wish!
... }
>>> items = calories.items()
>>> type(items)
<class 'dict_items'>
>>> hasattr(items, '__next__')
False
>>> hasattr(items, '__iter__')
True
```

What is returned by .items()?

A *dictionary view* object.

Quacks like a dictionary view if it supports three things:

- `len(view)` returns the number of items
- `view` is iterable
- `(key, value) in view` returns `True` if that pair is in the dictionary; else, `False`.

Iterable Views

A view is iterable, so you can use it in a for loop:

```
>>> for food, count in calories.items():
...     print("{:.<20s} {:<d} cal".format(food, count))
...
ice cream..... 15 cal
slice of bacon..... 43 cal
apple..... 95 cal
cheddar cheese..... 113 cal
```

Dynamically updates

A view dynamically updates, even if the source dictionary changes:

```
>>> items = calories.items()
>>> len(items)
4
>>> calories['orange'] = 50
>>> len(items)
5
>>> ('orange', 50) in items
True
```

Other methods

There are two other methods on dictionaries, called `.keys()` and `.values()`. They also return views.

```
>>> foods = calories.keys()
>>> counts = calories.values()
>>> 'yogurt' in foods
False
>>> 100 in counts
False
>>> calories['yogurt'] = 100
>>> 'yogurt' in foods
True
>>> 100 in counts
True
```

Benefits

Views improve over regular iterators:

- Are iterable, so can spawn multiple iterators
- Let you pass dict contents to caller and know it won't be modified
- Support extra services, like `len()` and `(key, val) in view`

And of course, views are more scalable & performant than a list of (key, value) pairs.

What about Python 2?

All the above was for Python 3. Here's how it works in 2:

- `calories.items()` returns a list of `(key, value)` tuples.
  - So if it has 100,000 entries...

- `iteritems()`: returns an iterator over the key-value tuples
- `viewitems()`: which returned a view

`iteritems` is basically obsoleted by `viewitems`, but most people don't realize this yet.

Obsolete methods

In Python 3, what used to be called `viewitems()` was renamed `items()`, and the old `items()` and `iteritems()` went away.

If you still need an actual list in Python 3, you can just say `list(calories.items())`