Elegant and Efficient Python

“The difference between the right word and the almost right word is the difference between lightning and a lightning bug.” --Mark Twain

Steve R. Hastings
steve@hastings.org

http://linuxfestnorthwest.org/sites/default/files/slides/elegant_and_efficient_python.odp

Last Edited: 2015-04-27

See Also the Presenter Notes

- I have added explanations of the various slides in the “presenter notes”. Look for them.

Python Hits the Sweet Spot

- Python is a fantastic combination of power and expressiveness

“Pythonic” Code

- Elegant code is “Pythonic”

Zen of Python and PEP 8

- The famous “Zen of Python”

  https://www.python.org/dev/peps/pep-0020/

  - Beautiful is better than ugly.
  - Explicit is better than implicit.
  - Simple is better than complex.
  - Complex is better than complicated.
  - ...[15 more; go read them for yourself!]
- Style guide: PEP 8

  https://www.python.org/dev/peps/pep-0008/

Learn Python Idioms

- The legendary “Code Like a Pythonista”

  http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html
- Recorded lectures by Raymond Hettinger
  - “Transforming Code into Beautiful, Idiomatic Python” (from PyCon 2013)

  https://www.youtube.com/watch?v=OSGv2VnC0go

The Code You Don't Write

- The fastest code is often terse and elegant

`for` Loops Without Indexing

- Don't do this:

```
colors = ['red', 'blue', 'green']

for i in range(len(colors)):
    print(colors[i])
```

- Do this:

```
for color in colors:
    print(color)
```

Use the Built-Ins 0

- Built-ins can often eliminate a loop
  - **`any()`**, **`all()`**, **`min()`**, **`max()`**, **`sum()`**

Use the Built-Ins 1

- Needless loop:

```
total = 0
for n in data:
   total += n

mean = total / len(data)
```

- Better:

```
mean = sum(data) / len(data)
```

Slicing Not Loops 0

- Average (mean) of first N elements

```
total = 0
for i in range(num_elements):
   total += data[i]
mean = total / num_elements


mean = sum(lst[:num_elements]) / num_elements
```

Slicing Not Loops 1

- Reversing by slicing

```
for color in colors[::-1]:
    print(color)
```

- Or, use the built-in!

```
for color in reversed(colors):
    print(color)
```

Use `iter()` to make an iterator

```
for i in my_list:
    print(i)
```

- This loop actually works like this internally:

```
my_iterator = iter(my_list)
while True:
    try:
        i = next(my_iterator)
        print(i)
    except StopIteration:
        break
```

Explicit Iterator Using `iter()`

```
my_iterator = iter(my_list)

for x in my_iterator:
    if x == 'special_header': break

count = int(next(my_iterator))

for x in my_iterator:
    count -= 1
    if count < 0: break
    my_function(x)
```

Jargon: Iterable

Open Files Are Iterators 0

- Elegantly skip first 3 lines

```
with open(my_file, 'r') as f:
    # First, throw away three lines
    for _ in range(3):
        next(f)
    # Next, do something with remaining lines
    for line in f:
        my_function(line)
```

Open Files Are Iterators 1

- Elegantly skip over a header

```
with open(my_file, 'r') as f:
    for line in f:
        if line.startswith("end_header"):
            break
    for line in f:
        my_function(line)
```

Bad: Special First Iteration

```
my_list = []

first_pass = True
with open(my_file, 'r') as f:
    for x in f:
        if first_pass:
            header = x
            first_pass = False
        else:
            my_list.append(x)
```

Better: Re-use Iterator

```
with open(my_file, 'r') as f:
    header = next(f)
    my_list = list(f)
```

Python Module: `itertools`

- Study it! Learn it!

`itertools` Example

```
from itertools import dropwhile, islice

def before_header(x):
    return x != 'special_header'

my_iterator = dropwhile(before_header, my_list)
next(my_iterator) # drop all through header

count = int(next(my_iterator))

for x in islice(my_iterator, count):
    my_function(x)
```

`lambda` for Throw-Away Functions

- A very simple function:

```
def before_header(x):
    return x != 'special_header'
```

- We can do this:

```
before_header = lambda x: x != 'special_header'
```

Itertools: **dropwhile** with **lambda**

```
import itertools as it

my_iterator = it.dropwhile(
        lambda x: x != 'special_header',
        my_list)
next(my_iterator) # drop all through header

count = int(next(my_iterator))

for x in it.islice(my_iterator, count):
    my_function(x)
```

`lambda` Isn't “Better”

- Don't do this:

```
cube = lambda x: x**3
```

- Do this:

```
def cube(x):
   return x**3
```

Jargon: “binding names”

```
# bind name x to new string object
x = "example"

# bind name y to same string object
y = x
print(x is y)  # prints True

# create new string object ending in '!'
x += "!"
print(x is y)  # now prints False
```

Jargon: Mutable

- Mutable example: list

```
my_list = []
my_list.append(1)  # we just mutated my_list
```

- Immutable example: string

```
s = "example"
s[0] = 'x'  # raises an exception
```

Command/Query Separation

```
x = my_list.sort()  # we just mutated my_list
assert x is None


id0 = id(my_list)
lst_sorted = sorted(my_list)
assert id0 != id(lst_sorted)  # new list object
```

Iterating and Mutating

- **Fails:**

```
for key in my_dictionary:
    if is_bad(key):
        del my_dictionary[key]
```

- **Works:**

```
for key in list(my_dictionary):
    if is_bad(key):
        del my_dictionary[key]
```

Dictionary Full of Counts 0

```
d = {}
for name in lst_names:
    try:
        d[name] += 1
    except KeyError:
        d[name] = 1
```

Dictionary Full of Counts 1

```
d = {}
for name in lst_names:
    if name not in d:
        d[name] = 0
    d[name] += 1
```

Dictionary Full of Counts 2

```
d = {}

for name in lst_names:
    d.setdefault(name, 0)
    d[name] += 1
```

Dictionary Full of Counts 3

```
d = {}

for name in lst_names:
    d[name] = d.get(name, 0) + 1
```

Dictionary Full of Counts 4

```
from collections import defaultdict

d = defaultdict(int)
for name in lst_names:
    d[name] += 1
```

Dictionary Full of Counts 5

```
from collections import Counter

d = Counter(lst_names)
```

List Comprehensions

```
data = [1.2, 2, 2.2, 3, 3.2, 4]

my_list = [x**2 for x in data if x==int(x)]
assert my_list == [4, 9, 16]
```

Dictionary Comprehensions

```
d = { chr(ord('a')+i) : i+1 for i in range(26)}

print(d['a'])  # prints 1
print(d['z'])  # prints 26
```

Set Comprehensions

```
a = {ch for ch in 'Python' if ch not in 'Pot'}

print(a)  # prints set(['y', 'n', 'h'])
```

Generator Expressions 0

```
data = [2, 2.5, 3, 3.5, 4]
g = (x**2 for x in data if x==int(x))

print(next(g))  # prints 4
print(next(g))  # prints 9
print(next(g))  # prints 16
print(next(g))  # raises StopIteration
```

Generator Expressions 1

```
from math import sqrt

rms = sqrt(sum(x**2 for x in data) / len(data))
```

Generator Expressions 2

```
with open(my_file, 'r') as f:
    result = sum(int(line) for line in f)
```

Generator Expressions 3

```
with open(my_file, 'r') as f:
    count = sum(1 for line in f if line.strip())
```

```
with open(my_file, 'r') as f:
    count = sum(line.strip() != '' for line in f)
```

Generator Expressions 4

```
bad_words = set(['gosh', 'heck', 'darn'])

my_list = 'You are a darn bad person'.split()
bad = any(w in bad_words for w in my_list)
assert bad == True

my_list = 'the Spanish Inquisition'.split()
bad = any(w in bad_words for w in my_list)
assert bad == False
```

Generator Expressions 5

```
bad_words = set(['gosh', 'heck', 'darn'])

with open(my_file, 'r') as f:
    bad = any(
        any(bad_word in word
                for bad_word in bad_words)
        for line in f for word in line.split())
```

Generator Expressions 6

- Do this:

```
total = sum(x**2 for x in iterable)
```

- Don't do this:

```
                X                                 X
total = sum([x**2 for x in iterable])
                X                                 X
```

Generators

```
def gen_range(start, stop):
    i = start
    while i < stop:
        yield i
        i += 1

g = gen_range(0, 3)
print(next(g)) # prints 0
print(next(g)) # prints 1
print(next(g)) # prints 2
print(next(g)) # raises StopIteration
```

Counting Words From a File

```
from collections import defaultdict

d = defaultdict(int)

with open(fname, 'r') as f:
    for line in f:
        for word in line.split():
            d[word] += 1
```

Generator: **`read_words()`**

```
def read_words(file_name):
    with open(file_name, 'r') as f:
        for line in f:
            for word in line.split():
                yield word
```

Using `read_words()`

```
from collections import Counter
counts = Counter(read_words(my_file))
```

```
bad_words = set(['gosh', 'heck', 'darn'])
bad = any(word in bad_words
          for word in read_words(my_file))
```

```
total = sum(int(word)
          for word in read_words(my_file)
```

Tree Traversal 0

```
class Node(object):
    def __init__(self, operator, left, right):
        self.operator = operator
        self.left = left
        self.right = right

def echo(x):
    print('{} '.format(x), end='')
```

Tree Traversal 1

```
(1 + 2) * (3 + 4)
```

```
expr = Node('*', Node('+', 1, 2), Node('+', 3, 4))
```

Tree Traversal 2

```
def prefix_echo(tree):
    echo(tree.operator)

    if hasattr(tree.left, 'operator'):
        prefix_echo(tree.left)
    else:
        echo(tree.left)

    if hasattr(tree.right, 'operator'):
        prefix_echo(tree.right)
    else:
        echo(tree.right)
```

Tree Traversal 3

```
prefix_echo(expr) # echoes: * + 1 2 + 3 4
```

Generators: Prefix Traversal

```
def prefix(tree):
    yield tree.operator

    if hasattr(tree.left, 'operator'):
        yield from prefix(tree.left)
    else:
        yield str(tree.left)

    if hasattr(tree.right, 'operator'):
        yield from prefix(tree.right)
    else:
        yield str(tree.right)
```

Generators: Infix Traversal

```
def infix(tree):
    if hasattr(tree.left, 'operator'):
        yield from infix(tree.left)
    else:
        yield str(tree.left)

    yield tree.operator

    if hasattr(tree.right, 'operator'):
        yield from infix(tree.right)
    else:
        yield str(tree.right)
```

Generators: Postfix Traversal

```
def postfix(tree):
    if hasattr(tree.left, 'operator'):
        yield from postfix(tree.left)
    else:
        yield str(tree.left)

    if hasattr(tree.right, 'operator'):
        yield from postfix(tree.right)
    else:
        yield str(tree.right)

    yield tree.operator
```

Traversal Is Abstracted

```
# original expression: (1 + 2) * (3 + 4)
# expr contains binary tree of above
print(' '.join(prefix(expr)))
# prints: * + 1 2 + 3 4
print(' '.join(infix(expr)))
# prints: 1 + 2 * 3 + 4
print(' '.join(postfix(expr)))
# prints: 1 2 + 3 4 + *
```

Traversal Without `yield from`

```
def prefix(tree):
    yield tree.operator

    if hasattr(tree.left, 'operator'):
        for x in prefix(tree.left): yield x
    else:
        yield str(tree.left)

    if hasattr(tree.right, 'operator'):
        for x in prefix(tree.right): yield x
    else:
        yield str(tree.right)
```

Context Managers 0

```
f = open(my_file, 'r')
try:
    for line in f:
        my_function(line)
finally:
    close(f)


with open(my_file, 'r') as f:
    for line in f:
        my_function(line)
```

Context Managers 1

```
try:
    os.remove(my_file)
except FileNotFoundError:
    pass
```

```
from contextlib import suppress

with suppress(FileNotFoundError):
    os.remove(my_file)
```

Python 2.x `OSError`

```
import errno, os

try:
    os.remove(my_file)
except OSError as e:
    if e.errno != errno.ENOENT:
        raise
    else:
        pass
```

Write Your Own Context Manager

```
class suppress_oserror(object):
    def __init__(self, errno):
        self.errno = errno

    def __enter__(self):
        return self

    def __exit__(self, e_type, e_val, e_tb):
        if e_type is not OSError:
            return False
        if e_val.errno != self.errno:
            return False
        return True
```

Using `suppress_oserror()`

```
import errno, os

with suppress_oserror(errno.ENOENT):
    os.remove(my_file)
```

Decorators

```
from some_module import some_decorator

def my_function(x):
    return x + 5
my_function = some_decorator(my_function)


@some_decorator
def my_function(x):
    return x + 5
```

contextlib.contextmanager

```
import contextlib, logging, time

@contextlib.contextmanager
def timed_block(level, mesg):
    # before yield is the 'enter' part
    start = time.time()
    yield
    # after yield is the 'exit' part
    stop = time.time()
    elapsed = stop - start
    logging.log(level, mesg +
                '{} secs'.format(elapsed))
```

Using `timed_block()`

```
from logging import INFO

with timed_block(INFO, 'block exec time'):
    my_function0()
    my_function1()
    my_function2()

# The time to run the block is logged.
```

Writing a Simple Decorator 0

- Disclaimer!
- There are plenty of blog posts and book examples showing the full details
- This is just the bare bones, for simplicity

Writing a Simple Decorator 1

```
def log_execution_time(fn):
    def my_wrapper(*args, **kwargs):
        start = time.time()
        result = fn(*args, **kwargs)
        stop = time.time()
        elapsed = stop - start
        log.info('{} took: {} seconds'.format(
                 fn.__name__, elapsed)
        return result
    return my_wrapper
```

Best Practice: `@wraps`

```
from functools import wraps

@wraps
def log_execution_time(fn):
    def my_wrapper(*args, **kwargs):
        start = time.time()
        result = fn(*args, **kwargs)
        stop = time.time()
        elapsed = stop - start
        log.info('{} took: {} seconds'.format(
                 fn.__name__, elapsed)
        return result
    return my_wrapper
```

Using the Simple Decorator

```
@log_execution_time
def my_function(x):
    return x + 5

result = my_function(5) # execution time logged
```

Making the Decorator Optional

```
import os

if os.getenv('LOG_EXEC_TIME', False):
    # use our decorator
    log_exec_time = log_execution_time
else:
    # set it to a do-nothing decorator
    def log_exec_time(fn):
        return fn
```

Thank You!

- Thank you for attending my talk. I hope you enjoyed it and/or learned something new.
- ZipRecruiter.com is hiring people with Python or Perl skills (including work-from-home)

  http://www.ziprecruiter.com/

ZipRecruiter®

Elegant and Efficient Python

“The difference between the right word and the almost right word is the difference between lightning and a lightning bug.” --Mark Twain

Steve R. Hastings
steve@hastings.org

http://linuxfestnorthwest.org/sites/default/files/slides/elegant_and_efficient_python.odp

Last Edited: 2015-04-27

- This presentation was made in LibreOffice Impress. You can download the presentation file from this URL:

  http://linuxfestnorthwest.org/sites/default/files/slides/elegant_and_efficient_python.odp

See Also the Presenter Notes

- I have added explanations of the various slides in the “presenter notes”. Look for them.

- Good job, you found the presenter notes!

Python Hits the Sweet Spot

- Python is a fantastic combination of power and expressiveness

- Python is a powerful, high-level language
- Python is clean and expressive; easy to read
- Python is "The Universal Language"
  - Useful for many purposes; “Batteries Included”
- Python's combination of power, expressiveness, and utility sets it apart
  - Python “fits my brain”

“Pythonic” Code

- Elegant code is “Pythonic”

- Python coders (“Pythonistas”) widely agree on many aspects of Python coding style
- Code that conforms to the general approved style is “Pythonic” code
- “Pythonic” also applies to commonly approved idioms
  - Pythonic idioms are usually very efficient
    - If the idioms were slow, who would use them?

Zen of Python and PEP 8

- The famous “Zen of Python”
  https://www.python.org/dev/peps/pep-0020/
  - Beautiful is better than ugly.
  - Explicit is better than implicit.
  - Simple is better than complex.
  - Complex is better than complicated.
  - ...[15 more; go read them for yourself!]
- Style guide: PEP 8
  https://www.python.org/dev/peps/pep-0008/

Learn Python Idioms

- The legendary “Code Like a Pythonista”

  http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html
- Recorded lectures by Raymond Hettinger
  - “Transforming Code into Beautiful, Idiomatic Python” (from PyCon 2013)

  https://www.youtube.com/watch?v=OSGv2VnC0go

The Code You Don't Write

- The fastest code is often terse and elegant

- Python, sadly, is about 50 times slower than C
- Python *programs* can still be fast!
- Make full use of the tools Python gives you
  - Both built-in language features and library modules
  - Library code can be fast C or even fast FORTRAN
    - It would be crazy to write your own matrix multiply; just use the one in NumPy.
- The fastest code is the code you didn't write!

`for` Loops Without Indexing

- Don't do this:

```
colors = ['red', 'blue', 'green']

for i in range(len(colors)):
    print(colors[i])
```

- Do this:

```
for color in colors:
    print(color)
```

Use the Built-Ins 0

- Built-ins can often eliminate a loop
  - **`any()`**, **`all()`**, **`min()`**, **`max()`**, **`sum()`**

Use the Built-Ins 1

- Needless loop:

```
total = 0
for n in data:
  total += n
mean = total / len(data)
```

- Better:

```
mean = sum(data) / len(data)
```

Slicing Not Loops 0

- Average (mean) of first N elements

```
total = 0
for i in range(num_elements):
  total += data[i]
mean = total / num_elements


mean = sum(lst[:num_elements]) / num_elements
```

- I should have put in a slide about basic slicing. Well, you can read about it here:

  http://stackoverflow.com/questions/509211/explain-pythons-slice-notation

Slicing Not Loops 1

- Reversing by slicing

```
for color in colors[::-1]:
    print(color)
```

- Or, use the built-in!

```
for color in reversed(colors):
    print(color)
```

Use `iter()` to make an iterator

```
for i in my_list:
    print(i)
```

- This loop actually works like this internally:

```
my_iterator = iter(my_list)
while True:
    try:
        i = next(my_iterator)
        print(i)
    except StopIteration:
        break
```

- The loop doesn't actually have the variable name "my_iterator" of course
- The key points here: an iterator object is advanced using its "next" method
- The actual next method name is: `.__next__()`

Explicit Iterator Using `iter()`

```
my_iterator = iter(my_list)

for x in my_iterator:
    if x == 'special_header': break

count = int(next(my_iterator))

for x in my_iterator:
    count -= 1
    if count < 0: break
    my_function(x)
```

- Suppose you need to scan for a value, then stop the scan and do something else
- This is a contrived example, but you might legitimately do something like this when you are, for example, parsing a file
- You could of course also do the loop like this:

```
for _ in range(count):
    x = next(my_iterator)
    my_function(x)
```

- The underscore variable name is a convention that signals that we don't care about the value of that variable. We must provide a variable in that position in the for statement, but we really just want the count; we don't use the values.

Jargon: Iterable

- Anything is “iterable” if we can pull values from it, one value at a time
- Usually we iterate with a `for` loop
  - But can also use `sum()` and etc.
  - Can use `next()` on an actual iterator
- If you can pass it to `iter()` it is iterable!

Open Files Are Iterators 0

- Elegantly skip first 3 lines

```
with open(my_file, 'r') as f:
    # First, throw away three lines
    for _ in range(3):
        next(f)
    # Next, do something with remaining lines
    for line in f:
        my_function(line)
```

- Instead of using `next()`, you can of course use the `.readline()` method, which will do the exact same thing.
- The important idea I'm trying to get across is that an open file object is an iterator, and that you can treat it the same as any other iterator.

Open Files Are Iterators 1

- Elegantly skip over a header

```
with open(my_file, 'r') as f:
    for line in f:
        if line.startswith("end_header"):
            break
    for line in f:
        my_function(line)
```

- The important point here is that you can break out of one loop, then start looping with a second `for` loop and it is picking up where the first one left off. It's the same thing as the “Explicit Iterators with `iter()`” slide, but we didn't need to call `iter()` because the open file object is already an iterator.

Bad: Special First Iteration

```
my_list = []

first_pass = True
with open(my_file, 'r') as f:
    for x in f:
        if first_pass:
            header = x
            first_pass = False
        else:
            my_list.append(x)
```

- Neither elegant nor efficient!
- Every pass through the loop must execute that `if` statement, and we have to manage a flag variable to keep track of which loop it is. Yuck.

Better: Re-use Iterator

```
with open(my_file, 'r') as f:
    header = next(f)
    my_list = list(f)
```

- These lines do the same job as all the code on the previous slide
- No need for that `first_pass` flag
- Pass any iterator to `list()` and it will construct a list containing the values returned by the iterator

Python Module: `itertools`

- Study it! Learn it!

- Python module full of elegant and efficient tools
- When you need something to be fast, look in itertools and see if there is anything you can use

`itertools` Example

```
from itertools import dropwhile, islice

def before_header(x):
    return x != 'special_header'

my_iterator = dropwhile(before_header, my_list)
next(my_iterator) # drop all through header

count = int(next(my_iterator))

for x in islice(my_iterator, count):
    my_function(x)
```

- The example from a few slides back can be rewritten using `itertools` and it's a bit cleaner.

`lambda` for Throw-Away Functions

- A very simple function:

```
def before_header(x):
    return x != 'special_header'
```

- We can do this:

```
before_header = lambda x: x != 'special_header'
```

- In the example, the function `before_header()` only exists to use with `itertools.dropwhile()`.
  - And it's a very simple function.
- We can make a trivial function, so simple it doesn't even have a name. We do this with the `lambda` keyword.
- In Python, `lambda` can only be used with a single expression returning a single value.
  - If you need a multi-line function, just define the function the usual way. It will have a name but that's not a problem.

Itertools: **`dropwhile`** with **`lambda`**

```
import itertools as it

my_iterator = it.dropwhile(
        lambda x: x != 'special_header',
        my_list)
next(my_iterator) # drop all through header

count = int(next(my_iterator))

for x in it.islice(my_iterator, count):
    my_function(x)
```

- Since `itertools.dropwhile()` needs us to pass it a callable object for the conditional, we make a function object to pass in. Since this function is extremely simple and we don't need to reuse it, it's a good candidate for `lambda`; note that the function is never given a name here.

`lambda` Isn't “Better”

- Don't do this:

```
cube = lambda x: x**3
```

- Do this:

```
def cube(x):
   return x**3
```

- I have seen lambda used for no reason
  - If the function needs a name, don't use `lambda`

Jargon: “binding names”

```
# bind name x to new string object
x = "example"

# bind name y to same string object
y = x
print(x is y)  # prints True

# create new string object ending in '!'
x += "!"
print(x is y)  # now prints False
```

- In Python, everything is an object
- Any name may have any object bound to it
- One object can be bound to multiple names
- Strings are immutable so the += operator creates a new string object and re-binds the name to the new object.

Jargon: Mutable

- Mutable example: list

```
my_list = []
my_list.append(1)  # we just mutated my_list
```

- Immutable example: string

```
s = "example"
s[0] = 'x'  # raises an exception
```

- If you can change the internal state of an object, then that object is a “mutable” object

Command/Query Separation

```
x = my_list.sort()  # we just mutated my_list
assert x is None


id0 = id(my_list)
lst_sorted = sorted(my_list)
assert id0 != id(lst_sorted)  # new list object
```

- This is a convention only, not enforced by the language. Nothing will stop you from breaking this convention.
- Python strictly obeys “Command/Query Separation” and your code should too
- If a function doesn't mutate its arguments, it may return a value
- If a function mutates one or more of its arguments, it should return `None`

- `list.sort()` mutates the list; it sorts in place
  - Therefore it returns `None`

- `sorted()` returns a sorted copy of the list
  - Therefore it doesn't mutate the list

Iterating and Mutating

- **Fails:**

```
for key in my_dictionary:
    if is_bad(key):
        del my_dictionary[key]
```

- **Works:**

```
for key in list(my_dictionary):
    if is_bad(key):
        del my_dictionary[key]
```

- Bad idea to mutate and iterate at the same time
- The first example fails; raises an exception `RuntimeError: dictionary changed size during iteration`
- The second example works because we create a new list of just the keys of the dictionary, and the list is not mutated when the dictionary is.

Dictionary Full of Counts 0

```
d = {}
for name in lst_names:
    try:
        d[name] += 1
    except KeyError:
        d[name] = 1
```

- This is the “Easier to Get Forgiveness Than Permission” technique: try to do something, catch the exception if it doesn't work.
- Not appropriate here. It's not elegant, and initially the dictionary is empty, so the exception will happen once for each name.
- The alternative to this is “Look Before You Leap”, check to see if a key has a value before trying to increment it. The next slide will show that.

Dictionary Full of Counts 1

```
d = {}
for name in lst_names:
    if name not in d:
        d[name] = 0
    d[name] += 1
```

- This is the most straightforward way to get counts in a dictionary.

Dictionary Full of Counts 2

```
d = {}

for name in lst_names:
    d.setdefault(name, 0)
    d[name] += 1
```

- The `.setdefault()` method sets a value if the name is not already set to a value.

Dictionary Full of Counts 3

```
d = {}
for name in lst_names:
    d[name] = d.get(name, 0) + 1
```

- The `.get()` method returns the current value associated with the given key; if you provide a second argument, that will be provided as the default value if the key is not yet set.

Dictionary Full of Counts 4

```
from collections import defaultdict

d = defaultdict(int)
for name in lst_names:
    d[name] += 1
```

- Python provides the elegant `defaultdict`
  - Pass a callable function that works with no argument
  - The callable will be called to create the default value

Dictionary Full of Counts 5

```
from collections import Counter

d = Counter(lst_names)
```

- “Batteries included”: `collections.Counter`
  - Solves this exact problem

- Moral of the story: learn the Python library
  - You may find something that solves your exact problem

List Comprehensions

```
data = [1.2, 2, 2.2, 3, 3.2, 4]

my_list = [x**2 for x in data if x==int(x)]
assert my_list == [4, 9, 16]
```

- List comprehensions provide a terse way to build a custom list
- Three parts: expression, `for` clause, `if` clause
  - Expression: how to compute each value
  - `for` clause: how to get values to use in expression
  - `if` clause: optional, filters some values out

Dictionary Comprehensions

```
d = { chr(ord('a')+i) : i+1 for i in range(26)}

print(d['a'])  # prints 1
print(d['z'])  # prints 26
```

- `ord(ch)` returns an integer representing the specified character, the “ordinal value” of that character. `ord('a')` will return 97 on any modern computer.
- `chr(n)` returns the letter (“character”) corresponding to the given ordinal value. `chr(97)` returns `'a'` on any modern computer.
- Thus this one line builds a dictionary mapping `'a'` to 1, `'b'` to 2, and so on through `'z'` mapping to 26.

Set Comprehensions

```
a = {ch for ch in 'Python' if ch not in 'Pot'}

print(a)  # prints set(['y', 'n', 'h'])
```

- Of course, with a set, the order of the letters printed might vary from what you see here. A set doesn't keep any particular ordering on its data.

Generator Expressions 0

```
data = [2, 2.5, 3, 3.5, 4]
g = (x**2 for x in data if x==int(x))

print(next(g))  # prints 4
print(next(g))  # prints 9
print(next(g))  # prints 16
print(next(g))  # raises StopIteration
```

- Syntax is nearly identical to listcomp
  - But uses parentheses instead of square brackets
  - If you use it in a function call, usually the function's parentheses work for the generator expression!
- Produces an iterator
  - Specifically: a generator object
- Use when you just want the value
  - Avoid building a list, using it once, then deleting it

Generator Expressions 1

```
from math import sqrt

rms = sqrt(sum(x**2 for x in data) / len(data))
```

- This example shows computing root mean square
- http://en.wikipedia.org/wiki/Root_mean_square

Generator Expressions 2

```
with open(my_file, 'r') as f:
    result = sum(int(line) for line in f)
```

- This example opens a text file, reads lines, converts each line to an integer, and sums the integers.
- This only works if there is one number per line on the text file.
- A later slide will show a similar example that can handle multiple numbers on a line.

Generator Expressions 3

```
with open(my_file, 'r') as f:
    count = sum(1 for line in f if line.strip())


with open(my_file, 'r') as f:
    count = sum(line.strip() != '' for line in f)
```

- Count the non-blank lines in a file
  - Count 1 for each non-blank line.

- Second example is the same thing, a trickier way.
  - Exploit that `bool` is a subclass of `int`
  - `True` has value 1, `False` has value 0
  - This seems a little sleazy to me but elegant at the same time! Can it be both?

Generator Expressions 4

```
bad_words = set(['gosh', 'heck', 'darn'])

my_list = 'You are a darn bad person'.split()
bad = any(w in bad_words for w in my_list)
assert bad == True

my_list = 'the Spanish Inquisition'.split()
bad = any(w in bad_words for w in my_list)
assert bad == False
```

- I apologize if anyone is shocked by the bad language on this example.
- Generator expressions are powerful and elegant when used in combination with `any()`, `all()`, `sum()`, `min()`, `max()` and so on.
- So, did you expect the Spanish Inquisition? Probably not... nobody expects the Spanish Inquisition.

Generator Expressions 5

```
bad_words = set(['gosh', 'heck', 'darn'])

with open(my_file, 'r') as f:
    bad = any(
        any(bad_word in word
                for bad_word in bad_words)
        for line in f for word in line.split())
```

- It is possible to overdo it with list comprehensions and generator expressions.
- This is just too much. It's messy and not elegant.
- In a few slides I'll show how to do this elegantly with a function (specifically, a generator function).

Generator Expressions 6

- Do this:

```
total = sum(x**2 for x in iterable)
```

- Don't do this:

```
              x                          x
total = sum([x**2 for x in iterable])
              x                          x
```

- The first example uses a generator expression to compute the needed values.
- The second one has square brackets, which means it is a list comprehension. It builds a list that is used once and then deleted. There's no need for that!
- People used to write code like that when Python didn't have generator expressions, but only had list comprehensions. Those days are over with.

Generators

```
def gen_range(start, stop):
    i = start
    while i < stop:
        yield i
        i += 1

g = gen_range(0, 3)
print(next(g)) # prints 0
print(next(g)) # prints 1
print(next(g)) # prints 2
print(next(g)) # raises StopIteration
```

- Generators allow you to think of how to generate a whole sequence, yet generate only one value at a time as needed.
- When you call a generator, you get back a generator object, like the object you get back from a generator expression.
  - A generator object is an iterator, and works like the other iterator examples already shown.

Counting Words From a File

```
from collections import defaultdict

d = defaultdict(int)

with open(fname, 'r') as f:
    for line in f:
        for word in line.split():
            d[word] += 1
```

- The example shows code for counting the words in a file.
- It is not elegant; file parsing is tangled up with the counting code.

Generator: **`read_words()`**

```
def read_words(file_name):
    with open(file_name, 'r') as f:
        for line in f:
            for word in line.split():
                yield word
```

- Word extraction is now available as a function. The function returns an iterator that works anywhere an iterable works in Python.

Using `read_words()`

```
from collections import Counter
counts = Counter(read_words(my_file))


bad_words = set(['gosh', 'heck', 'darn'])
bad = any(word in bad_words
        for word in read_words(my_file))


total = sum(int(word)
        for word in read_words(my_file)
```

- The generator completely decouples the parsing out of words from the text file, from what you do with the parsed words.
- Since the generator returns an iterator, we can simply use it like we used other iterators in other examples.
- The third example sums numbers from a file, and there can be multiple numbers per line.

Tree Traversal 0

```
class Node(object):
    def __init__(self, operator, left, right):
        self.operator = operator
        self.left = left
        self.right = right

def echo(x):
    print('{} '.format(x), end='')
```

- If you aren't familiar with tree traversal and this seems weird, feel free to skip this section.
- Here's `Node()`, a simple class for a binary tree
  - Used to represent simple math expressions
- Also, the `echo()` function, a convenient way to print a value, followed by a space, with no newline.
  - `echo()` is here just to save space on a later slide.

Tree Traversal 1

```
(1 + 2) * (3 + 4)


expr = Node('*', Node('+', 1, 2), Node('+', 3, 4))
```

- The example shows a simple math expression.
- Using the `Node()` class, we can build a binary tree that represents the math expression.
  - The name `expr` is short for “expression”
  - There are three nodes in the binary tree: one that expresses (1 + 2), one that expresses (3 + 4), and one that expresses multiplying the previous two. The binary tree makes the grouping totally unambiguous; the positions of the nodes in the tree strictly specify the order of the operations. No parentheses needed.

Tree Traversal 2

```
def prefix_echo(tree):
    echo(tree.operator)

    if hasattr(tree.left, 'operator'):
        prefix_echo(tree.left)
    else:
        echo(tree.left)

    if hasattr(tree.right, 'operator'):
        prefix_echo(tree.right)
    else:
        echo(tree.right)
```

- If you have never seen this before, it may seem tricky. But this can be explained very simply:
  - We always echo the operator first.
  - Then, we visit the left “child” of the node.
    - If the left has an “operator” member, it must be another node, so recursively call the function to visit that node.
    - If it doesn't have an “operator” member, it must be a number, so just echo the number.
  - Then, we visit the right “child” just as the left.
- The result: echo a prefix version of the expression.

Tree Traversal 3

```
prefix_echo(expr) # echoes: * + 1 2 + 3 4
```

- This shows what the output would be from running the code.

Generators: Prefix Traversal

```
def prefix(tree):
    yield tree.operator

    if hasattr(tree.left, 'operator'):
        yield from prefix(tree.left)
    else:
        yield str(tree.left)

    if hasattr(tree.right, 'operator'):
        yield from prefix(tree.right)
    else:
        yield str(tree.right)
```

- This is just like `prefix_echo()`, but instead of having calls to `echo()` hard-coded inside the function, it simply yields the values.
- Since all I want to do is print out the values, this converts the numbers to strings before yielding them.
  - A real program with a binary tree would probably return the values unchanged.

Generators: Infix Traversal

```
def infix(tree):
    if hasattr(tree.left, 'operator'):
        yield from infix(tree.left)
    else:
        yield str(tree.left)

    yield tree.operator

    if hasattr(tree.right, 'operator'):
        yield from infix(tree.right)
    else:
        yield str(tree.right)
```

- This is exactly like prefix(), but it does things in a different order. First visits left child, then yields the operator, then visits the right child.
- This returns in the familiar infix order.
- We could make a function that printed parentheses and then we would get back an expression that would evaluate properly. This version doesn't provide any parentheses.

Generators: Postfix Traversal

```
def postfix(tree):
    if hasattr(tree.left, 'operator'):
        yield from postfix(tree.left)
    else:
        yield str(tree.left)

    if hasattr(tree.right, 'operator'):
        yield from postfix(tree.right)
    else:
        yield str(tree.right)

    yield tree.operator
```

- Just like the previous two, but prints in postfix order. Visits left child, then visits right child, then yields up the operator.

Traversal Is Abstracted

```
# original expression: (1 + 2) * (3 + 4)
# expr contains binary tree of above
print(' '.join(prefix(expr)))
# prints: * + 1 2 + 3 4
print(' '.join(infix(expr)))
# prints: 1 + 2 * 3 + 4
print(' '.join(postfix(expr)))
# prints: 1 2 + 3 4 + *
```

- Prefix puts the operator first. This is unambiguous; no parentheses are needed to correctly evaluate the prefix expression.
- Infix is the familiar order; it needs parentheses to disambiguate order of operations.
- Postfix puts the operator after the values. This is unambiguous; no parentheses are needed to correctly evaluate the postfix expression.
  - In fact, if you have a “Reverse Polish Notation” calculator, you could type this in and it would produce the correct result.
- Each of these still represents the original expression; they are just different ways of looking at the expression.

Traversal Without `yield from`

```
def prefix(tree):
    yield tree.operator

    if hasattr(tree.left, 'operator'):
        for x in prefix(tree.left): yield x
    else:
        yield str(tree.left)

    if hasattr(tree.right, 'operator'):
        for x in prefix(tree.right): yield x
    else:
        yield str(tree.right)
```

Older versions of Python don't support `yield from`, so this is how you would implement a recursive function in a generator without `yield from`.

Context Managers 0

```
f = open(my_file, 'r')
try:
    for line in f:
        my_function(line)
finally:
    close(f)


with open(my_file, 'r') as f:
    for line in f:
        my_function(line)
```

- “Context Managers” set up a “context” for you
- Basically, they do setup and then do teardown
- Teardown occurs no matter what happens
  - Specifically, if an exception is raised, or not
- Some useful ones already built-in to Python
  - But you can write your own

- The first example shows the correct way to make sure a file is properly closed, even if `my_function()` raises an exception.
- The second example does the same thing, more conveniently, using a context manager.

Context Managers 1

```
try:
    os.remove(my_file)
except FileNotFoundError:
    pass



from contextlib import suppress

with suppress(FileNotFoundError):
    os.remove(my_file)
```

- Catching and suppressing the exception is the best practice for deleting a file.
- “Look Before You Leap” might not work, as there is a race condition: the file might exist when you test for its existence, and then be deleted right after the test. Then the file deletion would fail and the exception would be raised! Better to just try to delete the file, and handle the exception if it happens.

Python 2.x `OSError`

```
import errno, os

try:
    os.remove(my_file)
except OSError as e:
    if e.errno != errno.ENOENT:
        raise
    else:
        pass
```

- The code to correctly handle the exception is even more complicated in Python 2.x, because `os.remove()` just raises `OSError` for any error. You are expected to check the `errno` code to see specifically which error occurred.
- The next slide will show a context manager to suppress an `OSError` but only if the `errno` code matches a particular value.

Write Your Own Context Manager

```
class suppress_oserror(object):
    def __init__(self, errno):
        self.errno = errno

    def __enter__(self):
        return self

    def __exit__(self, e_type, e_val, e_tb):
        if e_type is not OSError:
            return False
        if e_val.errno != self.errno:
            return False
        return True
```

- Suppress Python 2.x os error by `errno`
  - In Python 2.x many os methods raise `OSError`
  - Suppress based on `.errno` attribute of exception object
- This may seem complex but it's not bad:
  - `__enter__()` sets up the context
  - `__exit__()` is called after the block completes, with information about the exception raised by the block (if any)
  - `__init__()` just sets up the context manager object.

Using `suppress_oserror()`

```
import errno, os

with suppress_oserror(errno.ENOENT):
    os.remove(my_file)
```

- In Python 2.x, this correctly supresses the `OSError` exception only when the `errno` is set to the “file does not exist” code.
- It would still allow the exception to be raised if the `errno` was some other value, such as `errno.EISDIR` “file is a directory”.

Decorators

```
from some_module import some_decorator

def my_function(x):
    return x + 5
my_function = some_decorator(my_function)


@some_decorator
def my_function(x):
    return x + 5
```

- The @decorator syntax “wraps” a function
  - The two examples have the same effect

contextlib.contextmanager

```
import contextlib, logging, time

@contextlib.contextmanager
def timed_block(level, mesg):
    # before yield is the 'enter' part
    start = time.time()
    yield
    # after yield is the 'exit' part
    stop = time.time()
    elapsed = stop - start
    logging.log(level, mesg +
            '{} secs'.format(elapsed))
```

- This shows how to make a simple context manager that measures how long the block takes to run, and writes a line into the log with that information.
- `contextlib.contextmanager` is a decorator. You use it to wrap your custom context manager function.
- Everything before the `yield` becomes the code that runs to set up the context; everything after the `yield` runs to finalize the context.

Using `timed_block()`

```
from logging import INFO

with timed_block(INFO, 'block exec time'):
    my_function0()
    my_function1()
    my_function2()

# The time to run the block is logged.
```

Writing a Simple Decorator 0

- Disclaimer!
- There are plenty of blog posts and book examples showing the full details
- This is just the bare bones, for simplicity

Writing a Simple Decorator 1

```
def log_execution_time(fn):
    def my_wrapper(*args, **kwargs):
        start = time.time()
        result = fn(*args, **kwargs)
        stop = time.time()
        elapsed = stop - start
        log.info('{} took: {} seconds'.format(
                fn.__name__, elapsed)
        return result
    return my_wrapper
```

- Create a new function object that includes the function to be wrapped
- Runs other code before the saved function, then runs the saved function, then runs more code after the saved function
- The `(*args, **kwargs)` saves up the arguments and keyword arguments, and then passes them when we call the saved function object.
- Finally we return the new function object that wraps the function.

Best Practice: `@wraps`

```
from functools import wraps

@wraps
def log_execution_time(fn):
    def my_wrapper(*args, **kwargs):
        start = time.time()
        result = fn(*args, **kwargs)
        stop = time.time()
        elapsed = stop - start
        log.info('{} took: {} seconds'.format(
                fn.__name__, elapsed)
        return result
    return my_wrapper
```

- Same as the previous slide except for the first two lines.
- The `@wraps` decorator makes sure that the new "wrapped" function returned by your decorator has the same name, plus the same docstring as the function being wrapped.

Using the Simple Decorator

```
@log_execution_time
def my_function(x):
    return x + 5

result = my_function(5) # execution time logged
```

Making the Decorator Optional

```
import os

if os.getenv('LOG_EXEC_TIME', False):
    # use our decorator
    log_exec_time = log_execution_time
else:
    # set it to a do-nothing decorator
    def log_exec_time(fn):
        return fn
```

- This shows how to make a decorator optional. Without changing the source code at all, you can enable or disable logging. Just set or unset the environment variable.
- The do-nothing decorator just gives the function object back, without actually wrapping it.
- You can use decorators to add possibly-slow tests or logging to your code... and if you disable them, there is *zero* cost in either time or memory. The functions are simply not wrapped so the possibly-slow code isn't even there to run!

Thank You!

- Thank you for attending my talk. I hope you enjoyed it and/or learned something new.
- ZipRecruiter.com is hiring people with Python or Perl skills (including work-from-home)

  http://www.ziprecruiter.com/

- If you have any questions about any of these slides, you can email me, or just post a question on StackOverflow.com and someone will help you.
- Thanks for reading through the slides from my talk!