Advanced Python I

by Raymond Hettinger
@raymondh
Files used in this tutorial

http://dl.dropbox.com/u/3967849/advpython.zip

Or in shortened form:

PSF board member
Python core developer since 2001
Author of the itertools module, set objects, sorting key functions, and many tools that you use every day
Consultant for training, code review, design and optimization
Background in data visualization and high-frequency trading
Person who is very interested in your success with Python
@raymondh on twitter
Background and Expectations

- What is the background of the participants?
- Who is beginner/intermediate moving to next level?
- Who is already somewhat advanced?
- What do you want to be able to do after the tutorial?
What does it mean to be Advanced?

- Know all of the basic language constructs and how they are used
- Understand the performance implications of various coding methods
- Understand how Python works
- Have seen various expert coding styles
- Actively use the docs as you code
- Know how to find and read source code
- Take advantage of programming in a dynamic language
- Become part of the Python community

In short, an advanced Python programmer becomes well-equipped to solve a variety of problems by fully exploiting the language and its many resources.
Foundation Skills for the Tutorial

- Accessing the documentation:
  - F1 on IDLE
  - Applications Directory or Window’s Start Menu
  - Doc/index.html on the class resource disk

- The interactive prompt:
  - IDLE, iPython, PyCharm, Wing-IDE, etc
  - command-line prompt with readline
  - PYTHONSTARTUP=~/pystartup.py  # tab-completion
  - Command line tools:
    python –m test.pystone
    python –m pdb
    python –m test.regrtest
Two Part Tutorial

- The morning will be full of techniques and examples designed to open your mind about Python’s capabilities.

- There will be periodic hands-on exercises

- The afternoon will have three parts:
  - All about Unicode
  - A sit back and relax presentation about descriptors
  - Guided exercises and problem sets
Foundation Skills for the Tutorial

- IDLE’s module loader
  - performs the same search as “import m”
  - fastest way to find relevant source no matter where it is
  - Mac users should map “Control Key M” to “open-module”

- IDLE’s class browser
  - hidden gem
  - fastest way to navigate unfamiliar source code
  - Control or Apple B

- Try it with the decimal module
Handy techniques for next section

- Bound methods are just like other callables:

```python
>>> s = []
>>> s_append = s.append
>>> s_append(3)
>>> s_append(5)
>>> s_append(7)
>>> s
[3, 5, 7]
```

- Accessing function names:

```python
>>> def fizzle(a, b, c):
...     ...

>>> fizzle.__name__
'fizzle'
```
Optimizations

- Replace global lookups with local lookups
  - Built-in names: list, int, string, ValueError
  - Module names: collections, copy, urllib
  - Global variables: even one that look like constants

- Use bound methods
  - \( bm = g.\text{foo} \)
  - \( bm(x) \) # same as \( g.\text{foo}(x) \)

- Minimize pure-python function calls inside a loop
  - A new stack frame is created on *every* call
  - Recursion is expensive in Python
Unoptimized Example

def one_third(x):
    return x / 3.0

def make_table(pairs):
    result = []
    for value in pairs:
        x = one_third(value)
        result.append(format(value, '9.5f'))
    return '
'.join(result)
def make_table(pairs):
    result = []
    result_append = result.append
    _format = format
    # bound method
    # localized
    for value in pairs:
        x = value / 3.0
        # in-lined
        result_append(_format(value, '9.5f'))
    return '\n'.join(result)
def dispatch(self, commands):
    for cmd in commands:
        cmd = {'duck': 'hide', 'shoot': 'fire'}.get(cmd, cmd)
        log(cmd)
        do(cmd)

def dispatch(self, commands):
    translate = {'duck': 'hide', 'shoot': 'fire'}
    for cmd in commands:
        cmd = translate.get(cmd, cmd)
        log(cmd)
        do(cmd)
Vectorization

- Replace CPython’s eval-loop with a C function that does all the work:

\[
\text{[ord(c) for c in long_string]}
\Rightarrow \text{list(map(ord, long_string))}
\]

\[
\text{[i**2 for i in range(100)]}
\Rightarrow \text{list(map(pow, count(0), repeat(2, 100))})
\]
if __name__ == '__main__':
    from timeit import Timer
    from random import random

    n = 10000
    pairs = [random() for i in range(n)]

    setup = "from __main__ import make_table, make_table2, pairs"
    for func in make_table, make_table2:
        stmt = '{0.__name__}(pairs)'.format(func)
        print(func.__name__, min(Timer(stmt, setup).repeat(7, 20)))
Class Exercise

- File: optimization.py
Goal Check

- Learn 5 techniques for optimization:
  - Vectorization
  - Localization
  - Bound Methods
  - Loop Invariant Code Motion
  - Reduce Function Calls

- Learn to measure performance with timeit.Timer()
  - See how the “import __main__” technique beats using strings

- Use func.__name__ in a loop

- Practice using itertools
Handy techniques for next section

- `pprint.pprint(nested_data_structure)`
- `help(pow)`
- `functools.partial()`

```python
>>> two_to = partial(pow, 2)
>>> two_to(5)
32
```
Think in terms of dictionaries

- **Files:** thinkdict/regular.py and thinkdict/dict_version.py

- **Experiments:**
  ```
  import collections
  vars(collections)
  dir(collections.OrderedDict)
  type(collections)
  dir(collections.Counter('abracadabra'))
  globals()
  help(instance)
  ```

- **Goal is to see dicts where other see modules, classes, instances, and other Python lifeforms**
Add a little polish

- Keyword arguments
- Docstrings
- Doctests
  doctest.testmod()
- Named tuples
  print(doctest.testmod())
ChainMap

- Common Pattern (but slow):

  ```python
def getvar(name, cmd_line_args, environ_vars, default_values):
    d = default_values.copy()
    d.update(environ)
    d.update(cmd_line_args)
    return d[name]
  ```

- Instead, link several dictionaries (or other mappings together for a quick single lookup):

  ```python
def getvar(name, cmd_line_args, environ_vars, default_values):
    d = ChainMap(cmd_line_args, environ_vars, default_values)
    return d[name]
  ```
Examples in Real Code

- Lib/string.py  # search for Template

- http://hg.python.org/cpython/file/default/Lib/configparser.py

- http://hg.python.org/cpython/file/default/Lib/collections.py
Goal Check

- Learn to see dictionaries where others see native python objects, classes, modules, etc.

- Develop an understanding of attribute and method lookup logic

- See how ChainMap() is used in real code
Who owns the dot?

- Take charge of the dot with `__getattribute__`

- Class demo: `own_the_dot/custom_getattribute`

- Basic Idea:
  Every time there is an attribute lookup
  Check the object found to see if it is an object of interest
  If so, invoke a method on that object
Class Exercise

Make a class with a custom `__getattribute__` that behaves normally, but logs each calls to stderr.
Goal Check

- Learn the underpinning of how descriptors are implemented
- Gain the ability to intercept attribute lookup and control the behavior of the dot.
- Deepen you understanding of attribute and method lookup logic
Exploiting Polymorphism

- Symbolic Expansion:
  \[ x + y \]
  where \( x \) and \( y \) are strings

- Example Files:
  - tracers/symbol_expansion.py
  - tracers/approximate.py

- Alternative to logging calls
Generating code

- Create code dynamically
  - Used when code can be parameterized or described succinctly

- Two ways to load
  - `exec()`
  - `import`

- Examples:
  - `collections.namedtuple()`
  - `codegen.py`
  - Ply introspects docstrings
Dynamic method discovery

- Framework technique that lets subclasses define new methods

- Dispatch to a given name is simple:

  ```python
  func = getattr(self, 'do_' + cmd)
  return func(arg)
  ```

- Given cmd=='move', this code makes a call to do_move(arg)
- See Lib/cmd.py at line 211
- See an example of a turtle shell in the cmd docs
Goal Check

- Learn how to evaluate functions symbolically

- Be able to generate code on the fly and load it with either exec() or an import.

- Know that docstrings can be used to guide code generation. Works well with a pattern->action style of coding.

- Be able to implement dynamic method discovery in a framework like cmd.py
Loops with Else-Clauses

def find(x, sequence):
    for i, x in enumerate(sequence):
        if x == target:
            # case where x is found
            break
        else:
            # target is not found
            i = -1
    return i
### Slicing

<table>
<thead>
<tr>
<th>Action</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Half-open interval: [2, 5)</td>
<td><code>s[2: 5]</code></td>
</tr>
<tr>
<td>Abbreviation for whole sequence</td>
<td><code>s[: ]</code></td>
</tr>
<tr>
<td>Copying a list</td>
<td><code>c = s[: ]</code></td>
</tr>
<tr>
<td>Clearing a list #1</td>
<td><code>del s[: ]</code></td>
</tr>
<tr>
<td>Clearing a list #2</td>
<td><code>s[: ] = []</code></td>
</tr>
</tbody>
</table>
# Negative Slicing

<table>
<thead>
<tr>
<th>Action</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Last element</td>
<td>s[-1]</td>
</tr>
<tr>
<td>Last two elements</td>
<td>s[-2 : ]</td>
</tr>
<tr>
<td>Two elements, one from the end</td>
<td>s[-3 : -1]</td>
</tr>
<tr>
<td>Empty slice</td>
<td>s[-2 : -2]</td>
</tr>
<tr>
<td>All the way back</td>
<td>‘abc’[-3]</td>
</tr>
<tr>
<td>Surprise wrap-around</td>
<td>for i in range(3): print 'abc'[::-i]</td>
</tr>
<tr>
<td></td>
<td>“”</td>
</tr>
<tr>
<td></td>
<td>‘ab’</td>
</tr>
<tr>
<td></td>
<td>‘a’</td>
</tr>
<tr>
<td></td>
<td>⇐ Empty!</td>
</tr>
</tbody>
</table>
Sorting skills

- See the sorting HowTo guide for details

- Key functions:
  - `key = str.upper`  # bound method
  - `key = lambda s: s.upper()`  # lambda
  - `key = itemgetter(2, 4)`  # third field and fifth field
  - `key = attrgetter('lastname', 'firstname')`
  - `key = locale.strxfrm()`

- SQL style with primary and secondary keys
Sorting skills

- Schwartzian transform:

  \[
  \text{decorated} = [(\text{func}(\text{record}), \text{record}) \text{ for } \text{record} \text{ in } \text{records}]
  \]
  \[
  \text{decorated}.\text{sort()}
  \]
  \[
  \text{result} = [\text{record} \text{ for } \text{key}, \text{record} \text{ in } \text{records}]
  \]

- Sort stability and multiple passes:

  \[
  \text{s}.\text{sort(key=attrgetter(\text{\textquoteleft last\textquoteright} \text{name}))} \quad \# \text{Secondary key}
  \]
  \[
  \text{s}.\text{sort(key=attrgetter(\text{\textquoteleft age\textquoteright}), reverse=True)} \quad \# \text{Primary key}
  \]
Goal Check

- Review Python basics with an eye towards mastery
- Loops with else-clauses
- Slicing invariants
- Handling of negative indices
- Sorting skills
Collections

- **Deque** – Fast $O(1)$ appends and pop from both ends
  
  ```python
d.append(10)          # add to right side
d.popleft()           # fetch from left side
  ```

- **Named Tuples** – Like regular tuples, but also allows access using named attributes
  
  ```python
  Point = namedtuple('Point', 'x y')
p = Point(10, 20)
  print p.x
  ```

- **Defaultdict** – Like a regular dictionary but supplies a factory function to fill-in missing values
  
  ```python
d = defaultdict(list)
d[k].append(v)          # new keys create new lists
  ```

- **Counter** – A dictionary that knows how to count
  
  ```python
c = Counter()
c[k] += 2               # zero value assumed for new key
  ```

- **OrderedDict** – A dictionary that remembers insertion order
LRU Cache

- Simple unbounded cache:
  ```python
def f(*args, cache={}):
    if args in cache:
        return cache[args]
    result = big_computation(*args)
    cache[args] = result
    return result
  ```

- But, that would grow without bound
- To limit its size, we need to throw-away least recently used entries
- Provided in the standard library as a decorator:
  ```python
  @functools.lru_cache(maxsize=100)
  def big_computation(*args):
    ...
Dynamic Programming with a Cache

```python
@lru_cache()
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(100))
```
Running trace from the command line

- python3.2 -m trace --count fibonacci.py

- Contents of fibonacci.cover:

```python
1: from functools import lru_cache

1: @lru_cache()
    def fibonacci(n):
101:     if n <= 1:
     2:         return n
99:     return fibonacci(n-1) + fibonacci(n-2)

1: print(fibonacci(100))
```
OrderedDict used to implement the LRU Cache

def f(*args, cache=OrderedDict()):
    if args in cache:
        result = cache[args]
        del cache[args]
    cache[args] = result
    return result
result = big_computation(*args)
cache[args] = result
if len(cache) > maxsize:
    cache.pop(0)
return result
Implicit Exception Chaining

try:
    1 / 0
except ZeroDivisionError:
    raise ValueError

Traceback (most recent call last):
    1 / 0
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
    raise ValueError
ValueError
Explicit Exception Chaining

try:
    1 / 0
except ZeroDivisionError as e:
    raise ValueError from e

Traceback (most recent call last):
    1 / 0
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
    raise ValueError
ValueError
Hierarchy issues with Exceptions

- Sometimes it is inconvenient that exceptions are arranged in a hierarchy
- We would sometimes like to be able to raise multiple kinds of exceptions all at once.
- The decimal module faces this challenge

class DivisionByZero(DecimalException, ZeroDivisionError):
class DivisionUndefined(InvalidOperation, ZeroDivisionError):
class Inexact(DecimalException):
class InvalidContext(InvalidOperation):
class Rounded(DecimalException):
class Subnormal(DecimalException):
class Overflow(Inexact, Rounded):
class Underflow(Inexact, Rounded, Subnormal):