Python
Tips and Tricks

by Raymond Hettinger
Python Core Developer

twitter: @raymondh

What's Ahead

- Unique Language Features and Idioms
- Comparisons and Ordering
- Programming with Iterators
- Programming Dynamically
- Collections
- Optimizations and Hacks
- The Debugger
- Development tools and Environment

Goal is for everyone to learn something new about programming Pythonically ☺

Unique Language Features and Idioms (some things are little different in Python)

- Slicing and Negative Indices
- Tuple packing and unpacking
- Iteration without indices
- Loops with else-clauses
- Ordering with key-functions

Slicing

Action	Code
Half-open interval: [2, 5)	s[2: 5]
Adding half-open intervals	s[2 :5] + s[5: 8] == s[2:8]
Abbreviation for whole sequence	s[:]
Copying a list	c = s[:]
Clearing a list #1	del s[:]
Clearing a list #2	s[:] = []

Negative Slicing

Action	Code
Last element	s[-1]
Last two elements	s[-2 :]
Two elements, one from the end	s[-3 : -1]
Empty slice	s[-2 : -2]
All the way back	‘abc’[-3]
Surprise wrap-around	for i in range(3): print 'abc'[:-i] ‘’ ← Empty! ‘ab’ ‘a’

Sequence Packing and Unpacking

Action	Code
pack into a tuple	t = 1, 2, 3
unpack from a tuple	a, b, c = t
packing and unpacking	a, b, c = 1, 2, 3
swapping values	a, b = b, a
evaluated RHS-to-LHS, left-to-right	i, a[i] = 1, 2 --- t = 1, 2 i = 1 a[1] = 2 New index !!!

No indices. The Pythonic Way™

Action	Code
Grow a list	s.append(x)
Shrink a list	s.pop(x)
Loop over a list	for x in s

Don't do this	Do this instead

for i in range(10):
  print s[i]
	for x in s:
	  print x

Iteration Idioms

Loop over	Code
numbers	for i in range(10)
sequence	for x in seq
backwards	for x in reversed(seq)
with indices	for i, x in enumerate(seq)
multiple sequences in lock-step	for x, y in zip(seq_x, seq_y)

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
```

skips else

run at end of sequence

Comparisons

- By default, all types are comparable
- None is always less than everything else

```
>>> sorted([3, None, 2])
[None, 2, 3]
```

- But complex() isn't orderable
- Rich comparisons allow each comparison operator to have a magic method

`x < y` uses `x.__lt__(y)`

- No guarantee that ordering is consistent
- Sets override comparison for subset/superset tests

```
>>> set('abc') < set('def')
False
```

Ordering Functions

sorted()	list.sort()
min()	max()
nsmallest()	nlargest()
bisect()	insort()

- These tools use __*lt*__ for comparisons
- The sort order is stable
- The first six support key functions

Key Functions

- Specifies a custom sort order
- Applies to each key no more than once
- Complex sort orders rely on multiple passes

sorted(s, key=str.lower)	Ignore case
sorted(s, key=lambda r: r[2])	Use the second field as the primary key
sorted(s, key=itemgetter(2))	Second field as primary key
sorted(s, key=lambda r: (r.last, r.first))	Lastname as primary key and Firstname as secondary key
sorted(s, key=attrgetter('last', 'first'))	Lastname primary, firstname secondary
sorted(s, key=attrgetter('dept')) sorted(s, key=attrgetter('age'), reverse=True))	Sort by department secondary, then descending age primary

Iterators are the Most Important Concept They make the language simple and clean

- What is iterable?
  - Strings, Lists, Tuples, Sets, Dictionaries, Deques, Files, URLs, generators, and any sequence with __getitem__, or any object with __iter__.

- What uses iterators?
  - For-loops, Set, Sorted, Min, Max, list comprehensions, generator expressions, dictionaries, etc.

- Examples

  sorted(set('abracadabra')) → '[a', 'b', 'c', 'd', 'r']

  dict(enumerate('abc')) → {0: 'a', 1: 'b', 2: 'c'}

Little known tool to make an Iterator

The iter() function has a two argument form that takes a function and a sentinel value

Example:

```
for block in iter(partial(f.read,20), ''):
    ...
```

Old way:

```
while True:
    block = f.read(20)
    if block == '':
        break
    ...
```

Not your father's super()

- In most languages, *super* is just a way to call a method on a parent class.
- In Python, *super* is a sophisticated tool designed to support co-operative multiple inheritance.
- In most languages, *super* is an immediate, fast reference to a parent class.
- In Python, *super* creates an instance of a custom class that overrides __getattribute__ and implements the descriptor protocol to achieve a call-time search of the MRO (method resolution order of the caller, not the original class).
- The search order is not knowable when you write a class. That is determined by subclasses.

A preview of what super() can do

- collections.Counter inherits from dict
- Semantics:
  - Any time Counter uses super(), it relies on another class to implement the actual mapping
  - By default, that other class is dict.
- Changing the delegate:

```
class OrderedCounter(Counter, OrderedDict):
    pass
```

- Now, Counter will delegate to OrderedDict before trying *dict*.

Collections Module

- Deque – Fast O(1) appends and pop from both ends

```
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

```
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

```
d = defaultdict(list)
d[k].append(v)      # new keys create new lists
```

- Counter – A dictionary that knows how to count

```
c = Counter()
c[k] += 2           # zero value assumed for new key
```

- OrderedDict – A dictionary that remembers insertion order

Tip:
Use DefaultDict for 2-D Sparse Arrays

Automatically creates the outermost dict entry:

```
d = defaultdict(dict)
d['Canada']['Quebec'] = 1
```

Old way:

```
d = dict()
d['Canada'] = dict()
d['Canada']['Quebec'] = 1
```

Programming Dynamically

- Use introspection
  - objects know their own type, their own methods, their own docstrings, sometimes known their name, and often have other meta-data attached
- No enums needed – use modules or classes instead
- Minimize multi-level subclassing (not the norm)
- Bound methods are not sinful
- Try and Except as control flow are not sinful either

Optimizations

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

- Use bound methods
  - bm = g.foo
  - bm(x) # same as g.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(seq):
    result = []
    for value in pairs:
        x = one_third(seq)
        result.append( format(value, '9.5f') )
    return '\n'.join(result)
```

Optimized Version

```
def make_table(seq):
    result = []
    result_append = result.append    # bound method
    _format = format                 # localized
    for value in seq:
        x = x / 3.0                  # in-lined
        result_append(_format(value, '9.5f'))
    return '\n'.join(result)
```

Little Known Hacks and Tricks (use them rarely and document them well)

Make a dictionary sparse

```
d.update(dict(d))   # doubles space in hash table
```

Turn-off thread-switching (cheap locking)

```
ci = getcheckinterval()
sys.setcheckinterval(sys.maxint)#switching  off
value = max(tasklist)
tasklist.remove(value)
sys.setcheckinterval(ci)          # switching on
```

Atomic actions (no locks required)

```
v = d.pop(key)      # find and remove in one-step
d.setdefault(key, []).append(v) # one-step init
```

PDB – The Python Debugger

- Automatic launch when exception is raised

  ```
  python -m pdb myscript.py
  ```

- Single Steps (step into and step over)
- Breakpoints
- Move up and down the stack frame
- List the script and mark currently executing line
- Run arbitrary commands including locals(), globals()
- Set new values in the middle of run
- Move up and down the stack frame

Sample PDB session

```
>>> import statistics
>>> import pdb
>>> pdb.run('statistics.sum([2,3,5])')
> <string>(1)<module>()
(Pdb) s
--Call--]
(Pdb) s
> /Users/raymond/Documents/
statistics.py(2)sum()
-> result = 0
(Pdb) s
> /Users/raymond/Documents/
statistics.py(3)sum()
-> for x in seq:
(Pdb) s
> /Users/raymond/Documents/
statistics.py(4)sum()
-> result += x
(Pdb) s
> /Users/raymond/Documents/
statistics.py(3)sum()
-> for x in seq:
(Pdb) s
> /Users/raymond/Documents/
statistics.py(4)sum()
-> result += x
(Pdb) l
  1         def sum(seq):
  2             result = 0
  3             for x in seq:
  4  ->             result += x
  5             return result
  6
[EOF]
(Pdb) !locals()
{'x': 3, 'result': 2, 'seq': [2, 3, 5]}
```

What is PDB good for?

- No more print statements to debug
- Automatic launch for an unhandled exception
- Can freeze the state of script in the middle of a run
- Let's you run diagnostic functions or change values on the fly ☺

Development Environment

Tools of the trade

- PIP and EasyInstall and PyPI
- VirtualEnv and VirtualEnv Wrapper
- Fabric
- IPython

PIP and EasyInstall and PyPI

- PyPI is giant, central listing of Python packages on python.org with links to each package
- PIP and Easy_Install are automatic package installers that pull from the PyPI links
- Easy_Install is well-known and widely used
- PIP downloads all dependencies *before* beginning the installation
- PIP can FREEZE all installed apps to a requirements file
- PIP can install from gzipped files or directly from version control repositories(svn, bzr, git, hg)

VirtualEnv and VirtualEnvWrapper

- Creates multiple, independent Python directory trees
- Allows simple experimentation with new versions of downloaded packages
- Allows separate builds with conflicting dependencies

```
$ mkvirtualenv my_experiment
$ workon my_experiment
$ pip install twisted
 . . .
$ deactivate
```

Fabric is a tool for remote deployment

Fabric like a distributed version of MAKE

Does one thing well: Bundle-up your local development environment, push it out to remote systems, and manage the restarts on remote system

Fabric Commands

- *local* – run a command on the local system
- *put* – copy from local to remote
- *run* – run a command on the remote file system

IPython

- Full featured Python shell
- Tab completion
- Great for experimentation
- Session logging and restore
- Remote and shared session
- Can run a script as an interactive demo

Popular Python Tools/Libraries

- Jython
  - Run Python on the JVM
  - Runs anywhere the JVM runs – everywhere
  - Automatic concurrency support – No GIL!
  - Full access to both Python and Java libraries

- IronPython
  - Runs on Microsoft's DotNet
  - Full access to dot net services

Popular Python Libraries

- Beautiful Soup
  - Builds a navigatable object tree from mal-formed HTML
- Twisted
  - Asynchronous toolkit. Build servers in minutes.
  - Uses a coroutine style instead of threads
  - Decision to Twisted is a deep, irreversible commitment!
- Django
  - Popular content management web framework
- NumPy and Matplotlib
  - Full suite of fast numeric tools and visualization tools

Questions and Answers

- twitter: @raymondh

More stuff to add

- 3-D sparse defaultdict
- from __builtin__ import *