Advanced Python Subjects

By Imri Goldberg
www.algorithm.co.il
plnnr.com

Introduction

- Many people I know come to Python from C/C++.
  - Including me!
- They bring with them many “unpythonic” idioms:
  - inheritance where it’s not needed
  - strong private methods
  - ...
- Other people aren’t yet aware of relatively new Python features
- We’ll start with basics and get to more advanced subjects from there
- Important:
  The idea is not to go out and start using everything **right now,** instead be aware of what’s available.

Other references

- Advanced Software Carpentry by Titus Brown
  - http://ivory.idyll.org/articles/advanced-swc/

__builtin__

- A good way to learn a lot, is to go over the list of builtin functions:
  - http://docs.python.org/library/functions.html
- Simple(st) example:
  C programmers do “for i in range(len(some_list)):” to iterate on two lists at once instead of using zip
- Interesting function list too long for a single slide, here are a few non-obvious examples:
  - any, all, getattr, reversed

stdlib

- Everyone knows about it, but there are many “hidden” gems
- Best introduction material to modules (besides docs) is “Python Module of the Week”:
  - http://www.doughellmann.com/PyMOTW/
- Examples include:
  - itertools, difflib, logging, ctypes, profile ...
- Don’t forget external modules as well:
  - numpy, scipy, coverage.py, etc...
  - See http://www.algorithm.co.il/blogs/index.php/programming/python/must-have-python-packages/
  - (shameless self promotion :)

IPython

- If you don’t know about it, google it right now, I’ll wait :)
- Best Python interactive prompt I know of today
- Includes:
  - completion, colors, shell, file editing, ...

Garbage Collection

- __del__ is somewhat like a destructor
- No guarantees when it will be called
- If your code has reference cycles + __del__:
  No cleanup!
- Avoid cycles with the weakref module
- Be aware of __del__ semantics

try-finally

- Don’t trust the gc to call __del__ when you don’t need the object anymore
- try-finally makes sure (within reason) that cleanup happens, predictably.
- Example:

-
  ```
  try:
      some_code()
  finally:
      cleanup()
  ```
- We’ll see another idiom building on try-finally later.

list comprehensions

- First “big leap” from C
- May be much more readable than other code:
  - `sum(x for x in some_list)`
- Simple examples:

  -
    ```
    In [1]: [x for x in range(5)]
    Out[1]: [0, 1, 2, 3, 4]
    ```

  -
    ```
    In [2]: [x**2 for x in range(10)]
    Out[2]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
    ```

  -
    ```
    In [3]: [x**2 for x in range(10) if x % 2 == 1]
    Out[3]: [1, 9, 25, 49, 81]
    ```

sets

- One of the more useful data structures
- Allows to quickly and efficiently compute intersections and unions without repetitions
- Examples:

  -
    ```
    In [1]: set(range(10)) - set(range(5))
    Out[1]: set([8, 9, 5, 6, 7])
    ```

  -
    ```
    In [2]: set([1,2,3]) & set([2,3,4])
    Out[2]: set([2, 3])
    ```

  -
    ```
    In [3]: set([1,2,3]) | set([2,3,4])
    Out[3]: set([1, 2, 3, 4])
    ```
- Use frozenset if you need to use sets as keys for a dict, or as items in another set.

Operator Overloading

- You can override standard operations
- The most known “special methods”:
  - __init__, __str__, __unicode__.
- Simple example:

```
class MyComplex(object):
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag
    def __add__(self, other):
        return MyComplex(self.real + other.real,
        self.imag + other.imag)
```

- Much more useful is overriding __getitem__ and __setitem__ (operator[]) to write your own container objects
- For example (using __contains__:
  - `some_user  in some_user_group`
- For the full list:
  - http://docs.python.org/reference/datamodel.html#special-method-names

Generators & Generator Expressions

- Generator are like list comprehensions, except that the values are only created when needed:

```
def p(x):
        print x
        return x

In [13]: [p(x) for x in range(5)]
0
1
2
3
4
Out[13]: [0, 1, 2, 3, 4]

In [14]: (p(x) for x in range(5))
Out[14]: <generator object at 0x02A24418>
```

Generators & Generator Expressions, cont.

```
In [15]: r = (p(x) for x in range(5))

In [16]: r.next()
0
Out[16]: 0

In [17]: [x**2 for x in r]
1
2
3
4
Out[17]: [1, 4, 9, 16]
```

- Notice when were the elements printed

Generators & Generator Exrpessions, Cont.

- You can write functions that behave similarly:

```
In [3]: def f():
   ...:     print 'before'
   ...:     yield 1
   ...:     print 'during'
   ...:     yield 2
   ...:     print 'after'
In [4]: r = f()
In [5]: r.next()
before
Out[5]: 1
In [6]: r.next()
during
Out[6]: 2
In [7]: r.next()
after
#Exception: StopIteration
```

Generators & Generator Expressions, Cont.

- What are generators good for:
  - Cleaner code,
  - Faster code,
  - Better idioms
  - Newer idioms (we'll see later on)
- Further reading:
  - http://www.dabeaz.com/generators/index.html (the presentation slides)

Closures

- Should be familiar to anyone coming from JS or lisp.
- What does the following function do?

-
  ```
  def f(x):
      def g(y):
          return x*y
      return g
  ```
- A closure is a function defined within another function, which accesses its surrounding function’s variables.
- G “saves” a copy of f’s state.
- function, which accesses it’s caller’s variables.
- What is it good for?

Decorators

- Syntactic sugar:

```
@decorator
def f(x):
    pass
```

$\Rightarrow$

```
def f(x):
    pass
f = decorator(f)
```

- Classic example:

```
def trace(func):
    def wrapper(*args):
        print '%s%s called' % (func.__name__, repr(args))
        ret = func(*args)
        print '%s -> %s' % (func.__name__, ret)
        return ret
    return wrapper
```

Decorators, cont.

- Example usage:

```
In [23]: @trace
   ....: def f(x):
   ....:     return x**2
   ....:

In [24]: f(1)
f(1,) called
f -> 1
Out[24]: 1
```

Properties

- "Hooks" for member access
- Allow clean transition from prototype code to clean encapsulated members

```
class A(object):
      def __init__(self):
            self._x = 0
      def _get_x(self):
            return self._x
      def _set_x(self, new_x):
            self._x = new_x
      x = property(_get_x, _set_x)
In [27]: a = A()
In [28]: a.x
Out[28]: 0
In [29]: a.x = 1
In [30]: a.x
Out[30]: 1
```

Properties, cont.

- Cute read-only property trick using decorators:

```
class B(object):
        def __init__(self, y):
                self._y = y
        @property
        def y(self):
                return self._y
In [35]: b = B(1)

In [36]: b.y
Out[36]: 1
```

with-statement

- A cleaner version for the following pattern:

```
obj = allocate()
try:
    do_something()
finally:
    obj.release()
```

→

```
with allocate() as obj:
    do_something()
```

- May be used on files, synchronization locks
- What if you want to write your own?

Writing your own context managers

- Override __enter__ and __exit__
- An easier way: use contextlib!
- A simple example:

```
@contextmanager
def tag(name):
    print "<%s>" % name
    yield
    print "</%s>" % name

>>> with tag("h1"):
...     print "foo"
...
<h1>
foo
</h1>
```

Writing your own context managers, cont.

- Continuing our example:

```
@contextlib.contextmanager
def allocate_foo():
        foo = allocate()
        try:
                yield foo
        finally:
                foo.release()

with allocate_foo() as foo:
              do_something()
```

Discussion

- Python is big, much to know
- You don’t know about a lot of things until you need them.
- Many more advanced subjects:
  - optimizations
  - writing C for Python
  - metaclasses
  - ...

- Anything you want to add?