

# Python Tips and Tricks

Stéphane Vialette

LIGM, Université Paris-Est Marne-la-Vallée

November 14, 2011

# Lists

## Checking a Condition on Any or Every List Element

Say you want to check to see if any element in a list satisfies a condition (say, it's below 10).

### Python

```
numbers = [1,10,100,1000,10000]
if [number for number in numbers if number < 10]:
    print 'At least one element is over 10'
# Output: 'At least one element is over 10'
```

# Lists

## Python

```
numbers = [1,10,100,1000,10000]
if [number for number in numbers if number < 10]:
    print 'At least one element is over 10'
# Output: 'At least one element is over 10'
```

## Why?

- If none of the elements satisfy the condition, the list comprehension will create an empty list which evaluates as false.
- Otherwise, a non-empty list will be created, which evaluates as true.
- Strictly, you don't need to evaluate every item in the list; you could bail after the first item that satisfies the condition.

# Lists

## Python

```
numbers = [1,10,100,1000,10000]
if any(number < 10 for number in numbers):
    print 'Success'
# Output: 'Success!'
```

## Why?

- With the new built-in `any` function introduced in Python 2.5, you can do the same thing cleanly and efficiently. `any` is actually smart enough to bail and return `True` after the first item that satisfies the condition.
- Here, we use a generator expression that returns a `True` or `False` value for each element, and pass it to `any`.
- The generator expression only computes these values as they are needed, and `any` only requests the values it need

# Lists

## Checking a Condition on Any or Every List Element

Say you want to check to see if every element in a list satisfies a condition (say, it's below 10).

### Python

```
numbers = [1,2,3,4,5,6,7,8,9]
if len(numbers) == len([number for number in numbers
                        if number < 10]):
    print 'Success!'
# Output: 'Success!'
```

# Lists

## Python

```
numbers = [1,2,3,4,5,6,7,8,9]
if len(numbers) == len([number for number in numbers
                        if number < 10]):
    print 'Success!'
# Output: 'Success!'
```

## Why?

- We filter with a list comprehension and check to see if we still have as many elements.
- If we do, then all of the elements satisfied the condition.
- Again, this is less efficient than it could be, because there is no need to keep checking after the first element that doesn't satisfy the condition.

# Lists

Python

```
numbers = [1,2,3,4,5,6,7,8,9]
if all(number < 10 for number in numbers):
    print 'Success!'
# Output: 'Success!'
```

Why?

- all smart enough to bail after the first element that doesn't match, returning False.
- This method works just like the any method described above.

# Lists

## Enumerate

- Remember (or maybe not) when you programmed in C, and for loops counted through index numbers instead of elements?
- Python has a really awesome built-in function called `enumerate` that will give you both. Enumerate-ing a list will return an iterator of index, value pairs

## Python

```
strings = ['a', 'b', 'c', 'd', 'e']  
for index, string in enumerate(strings):  
    print index, string,  
# prints '0 a 1 b 2 c 3 d 4 e'
```



# Lists

## Nested 'for' Statements

A python neophyte might write something like

Python

```
for x in (0,1,2,3):  
    for y in (0,1,2,3):  
        if x < y:  
            print (x, y, x*y),  
  
# prints (0, 1, 0) (0, 2, 0) (0, 3, 0) (1, 2, 2) (1, 3, 3)  
# (2, 3, 6)
```

# Lists

## Nested 'for' Statements

With a list comprehension, though, you can do this more quickly

### Python

```
print [(x, y, x * y) for x in (0,1,2,3)
        for y in (0,1,2,3)
        if x < y]
# prints [(0, 1, 0), (0, 2, 0), (0, 3, 0), (1, 2, 2),
# (1, 3, 3), (2, 3, 6)]
```

# Lists

Advanced logic with sets

Make sure a list is unique.

Python

```
numbers = [1,2,3,3,4,1]
set(numbers)
# returns set([1,2,3,4])

if len(numbers) == len(set(numbers)):
    print 'List is unique!'
# In this case, doesn't print anything
```

# Lists

Advanced logic with sets

Remove duplicates.

Python

```
numbers = [1,2,3,3,4,1]
numbers = list(set(numbers))
# returns [1,2,3,4] (or some other permutation of [1,2,3,4])
```

# Selecting values

The right way

The right way to select values inline

Python

```
test = True
# test = False
result = 'Test is True' if test else 'Test is False'
# result is now 'Test is True'
```

Python

```
test1 = False
test2 = True
result = 'Test1 is True' if test1 else 'Test1 is False, test2 is True' \
        if test2 else 'Test1 and Test2 are both False'
```

# Selecting values

## The and/or trick

- 'and' returns the first false value, or the last value if all are true. In other words, if the first value is false it is returned, otherwise the last value is returned.
- 'or' returns the first true value, or the last value if all are false.

## Python

```
test = True
# test = False
result = test and 'Test is True' or 'Test is False'
# result is now 'Test is True'
```

# Selecting values

## Python

```
test = True
# test = False
result = test and 'Test is True' or 'Test is False'
# result is now 'Test is True'
```

## How does this work?

- If test is true, the and statement skips over it and returns its right half, here 'Test is True' or 'Test is False'. As processing continues left to right, the or statement returns the first true value, 'Test is True'.
- If test is false, the and statement returns test. As processing continues left to right, the remaining statement is test or 'Test is False'. Since test is false, the or statement skips over it and returns its right half, 'Test is False'.

# Functions

Default argument values are only evaluated once

Here's a problem that has confused many new Python writers, including myself, repeatedly, even after I figured out the problem ...

Python

```
def function(item, stuff = []):  
    stuff.append(item)  
    print stuff
```

```
function(1)  
# prints '[1]'
```

```
function(2)  
# prints '[1,2]' !!!
```



# Functions

Default argument values are only evaluated once

- The solution: don't use mutable objects as function defaults.
- You might be able to get away with it if you don't modify them, but it's still not a good idea.

## Python

```
def function(item, stuff = None):  
    if stuff is None:  
        stuff = []  
    stuff.append(item)  
    print stuff
```

```
function(1)  
# prints '[1]'
```

```
function(2)  
# prints '[2]', as expected
```

# Functions

Default argument values are only evaluated once

You can forcefully re-evaluate the default arguments before each function call

## Python

```
from copy import deepcopy

def resetDefaults(f):
    defaults = f.func_defaults
    def resetter(*args, **kwds):
        f.func_defaults = deepcopy(defaults)
        return f(*args, **kwds)
    resetter.__name__ = f.__name__
    return resetter
```

# Functions

Default argument values are only evaluated once

You can forcefully re-evaluate the default arguments before each function call

## Python

```
@resetDefaults # a decorator
def function(item, stuff = []):
    stuff.append(item)
    print stuff

function(1)
# prints '[1]'

function(2)
# prints '[2]', as expected
```

# Functions

Default argument values are only evaluated once

You can forcefully re-evaluate the default arguments before each function call

## Python

```
@resetDefaults # a decorator
def function(item, stuff = []):
    stuff.append(item)
    print stuff

function(1)
# prints '[1]'

function(2)
# prints '[2]', as expected
```

# Functions

## Passing a list as arguments

Since you can receive arguments as a list or dictionary, it's not terribly surprising, I suppose, that you can send arguments to a function from a list.

## Python

```
args = [5,2]
pow(*args)
# returns pow(5,2), meaning 5^2 which is 25
```

# Classes

Checking for property and method existence

Need to know if a particular class or instance has a particular property or method?

Python

```
class Class:
    answer = 42

hasattr(Class, 'answer')
# returns True
hasattr(Class, 'question')
# returns False
```

# Classes

Checking for property and method existence

You can also check for existence of and access the property in one step using the built-in function 'getattr'.

Python

```
class Class:
    answer = 42

getattr(Class, 'answer')
# returns 42

getattr(Class, 'question', 'What is six times nine?')
# returns 'What is six times nine?'

getattr(Class, 'question')
# raises AttributeError
```

# Classes

## Modifying classes after creation

You can add, modify, or delete a class property or method long after the class has been created, and even after it has been instantiated.

## Python

```
class Class:
    def method(self):
        print 'Hey a method'

instance = Class()
instance.method()
# prints 'Hey a method'

def new_method(self):
    print 'New method wins!'

Class.method = new_method
instance.method()
# prints 'New method wins!'
```



# Classes

## Creating class methods

- A 'class method' receives the class as the first argument, just as a regular instance method receives the instance as the first argument.
- A 'static method' receives no information about where it is called; it is essentially a regular function, just in a different scope.

## Python

```
class Class:
    @classmethod
    def a_class_method(cls):
        print 'I was called from class %s' % cls
    @staticmethod
    def a_static_method():
        print 'I have no idea where I was called from'
    def an_instance_method(self):
        print 'I was called from the instance %s' % self
```

# Classes

## Python

```
class Class:
    @classmethod
    def a_class_method(cls):
        print 'I was called from class %s' % cls
    @staticmethod
    def a_static_method():
        print 'I have no idea where I was called from'
    def an_instance_method(self):
        print 'I was called from the instance %s' % self

instance = Class()
Class.a_class_method()
instance.a_class_method()
# both print 'I was called from class __main__.Class'
Class.a_static_method()
instance.a_static_method()
# both print 'I have no idea where I was called from'
Class.an_instance_method()
# raises TypeError
instance.an_instance_method()
# prints 'I was called from the instance <__main__.Class instance at 0x2e80d0>'
```

# Classes

## Shortcut for objects

All you want to do is create an object that holds data in several fields

## Python

```
class Struct:
    def __init__(self, **entries):
        self.__dict__.update(entries)

>>> options = Struct(answer=42, linelen = 80, font='courier')
>>> options.answer
42
>>> options.answer = 'plastics'
>>> vars(options)
{'answer': 'plastics', 'font': 'courier', 'linelen': 80}
```

# Classes

## Shortcut for objects

All you want to do is create an object that holds data in several fields

## Python

```
class Struct:
    def __init__(self, **entries):
        self.__dict__.update(entries)

    def __repr__(self):
        args = ['%s=%s' % (k, repr(v)) for (k,v) in vars(self).items()]
        return 'Struct(%s)' % ', '.join(args)

>>> options = Struct(answer=42, linelen = 80, font='courier')
>>> options
Struct(answer='plastics', font='courier', linelen=80)
```

## Can you implement abstract classes in Python?

- Java has an `abstract` keyword so you can define abstract classes that cannot be instantiated, but can be subclassed if you implement all the abstract methods in the class.
- It is a little known fact that you can use `abstract` in Python in almost the same way; the difference is that you get an error at runtime when you try to call the unimplemented method, rather than at compile time.

# Classes

Python

```
class MyAbstractClass:
    def method1(self): abstract

class MyClass(MyAbstractClass):
    pass

>>> MyClass().method1()
Traceback (most recent call last):
    ...
NameError: name 'abstract' is not define
```

# Classes

Can you implement abstract classes in Python?

If you're willing to write `abstract()` instead of `abstract`, then you can define a function that raises a `NotImplementedError` instead of a `NameError`, which makes more sense.

## Python

```
def abstract():
    import inspect
    caller = inspect.getouterframes(inspect.currentframe())[1][3]
    raise NotImplementedError(caller + ' must be implemented in subclass')

>>> MyDerivedClass().method1()
Traceback (most recent call last):
...
NotImplementedError: method1 must be implemented in subclass
```

# Classes

Can you implement abstract classes in Python?

Another solution using `__class__`.

Python

```
class AbstractClass(object):  
    def __init__(self):  
        if self.__class__ is AbstractClass:  
            raise NotImplementedError
```



# Strings

## Advices

- Triple quotes are an easy way to define a string with both single and double quotes.
- String concatenation is expensive. Use percent formatting and `join()` for concatenation.

## Python

```
print "Spam" + " eggs" + " and" + " spam"
print " ".join(["Spam", "eggs", "and", "spam"])

print "%s %s %s %s" % ("Spam", "eggs", "and", "spam")
```

*# DON'T DO THIS  
# Much faster/more  
# common Python idiom  
# Also a pythonic way of  
# doing it - very fast*

# Module

## Module choice

- cPickle is a faster, C written module for pickle.
- cPickle is used to serialize python program.
- Other modules have C implementations as well, cStringIO for the StringIO module, and cProfile for the profile module.

## Python

```
try:
    import cPickle as pickle
except ImportError:
    import pickle
```

# Data type choice

## context

Choosing the correct data type can be critical to the performance of an application.

## Python

*# Say you have 2 lists:*

```
list1 = [{'a': 1, 'b': 2}, {'c': 3, 'd': 4}, {'e': 5, 'f': 6}]
```

```
list2 = [{'e': 5, 'f': 6}, {'g': 7, 'h': 8}, {'i': 9, 'j': 10}]
```

*# find the entries common to both lists.*

```
common = []
```

```
for entry in list1:
```

```
    if entry in list2:
```

```
        common.append(entry)
```

# Data type choice

## context

Choosing the correct data type can be critical to the performance of an application.

## Python

*# Say you have 2 lists:*

```
list1 = [{'a': 1, 'b': 2}, {'c': 3, 'd': 4}, {'e': 5, 'f': 6}]
```

```
list2 = [{'e': 5, 'f': 6}, {'g': 7, 'h': 8}, {'i': 9, 'j': 10}]
```

*# find the entries common to both lists.*

```
set1 = set([tuple(entry.items()) for entry in list1])
```

```
set2 = set([tuple(entry.items()) for entry in list2])
```

```
common = set1.intersection(set2)
```

```
common = [dict(entry) for entry in common]
```