Pythonic Programming

Taking advantage of Python's strengths

Idiom

- In English:
  - "how come" (why)
  - "a piece of cake" (it's easy)
- Many things that are “normal” in other programming languages are anti-patterns in Python.

Batteries included

- Chances are that somebody has encountered (and solved) at least part of your problem
- Before you write any code, check:
  - Python standard library
  - PyPI
  - Python cookbook
  - Stack overflow
  - Google
  - ...

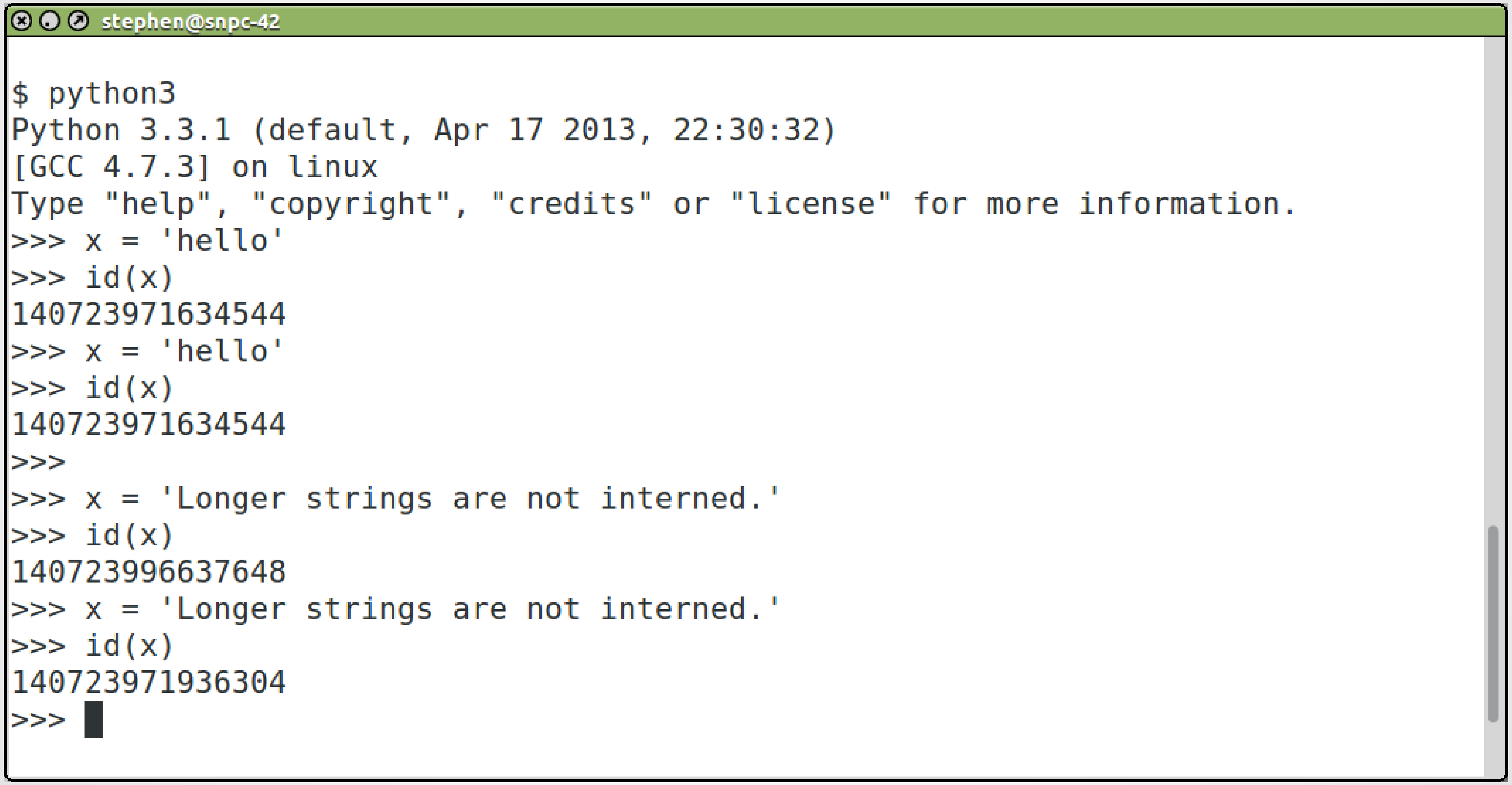
Things to know about Python

- Everything is a object
  - Some are mutable, some are immutable
- Most work is done at run time
  - Know what kind of object you're working with

Anti-patterns

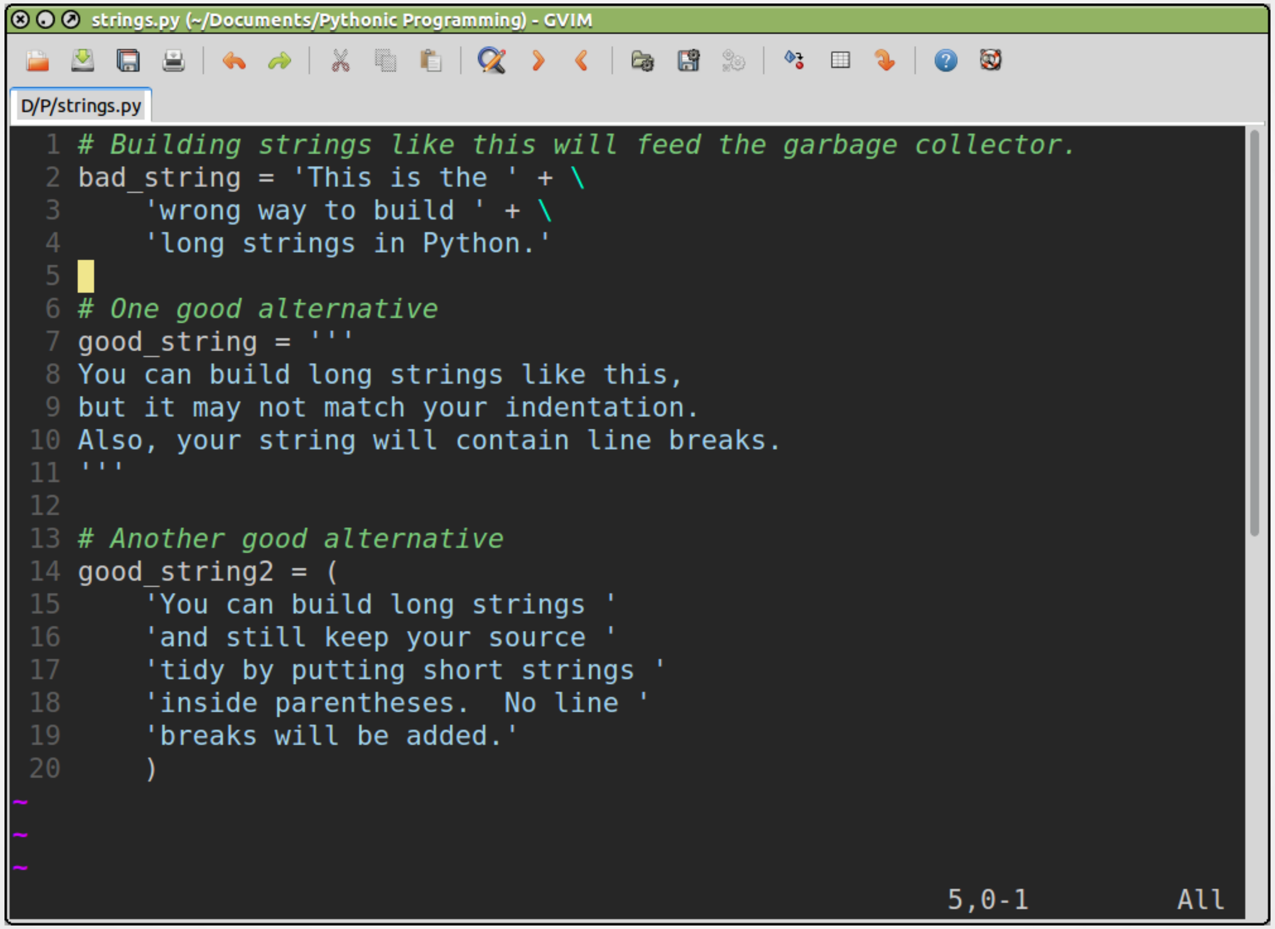
- Things that are acceptable in other languages are a bad idea in Python

Strings are immutable; don't be the Cookie Monster


stephen@snpc-42

```
$ python3
Python 3.3.1 (default, Apr 17 2013, 22:30:32)
[GCC 4.7.3] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> x = 'hello'
>>> id(x)
140723971634544
>>> x = 'hello'
>>> id(x)
140723971634544
>>>
>>> x = 'Longer strings are not interned.'
>>> id(x)
140723996637648
>>> x = 'Longer strings are not interned.'
>>> id(x)
140723971936304
>>> 
```

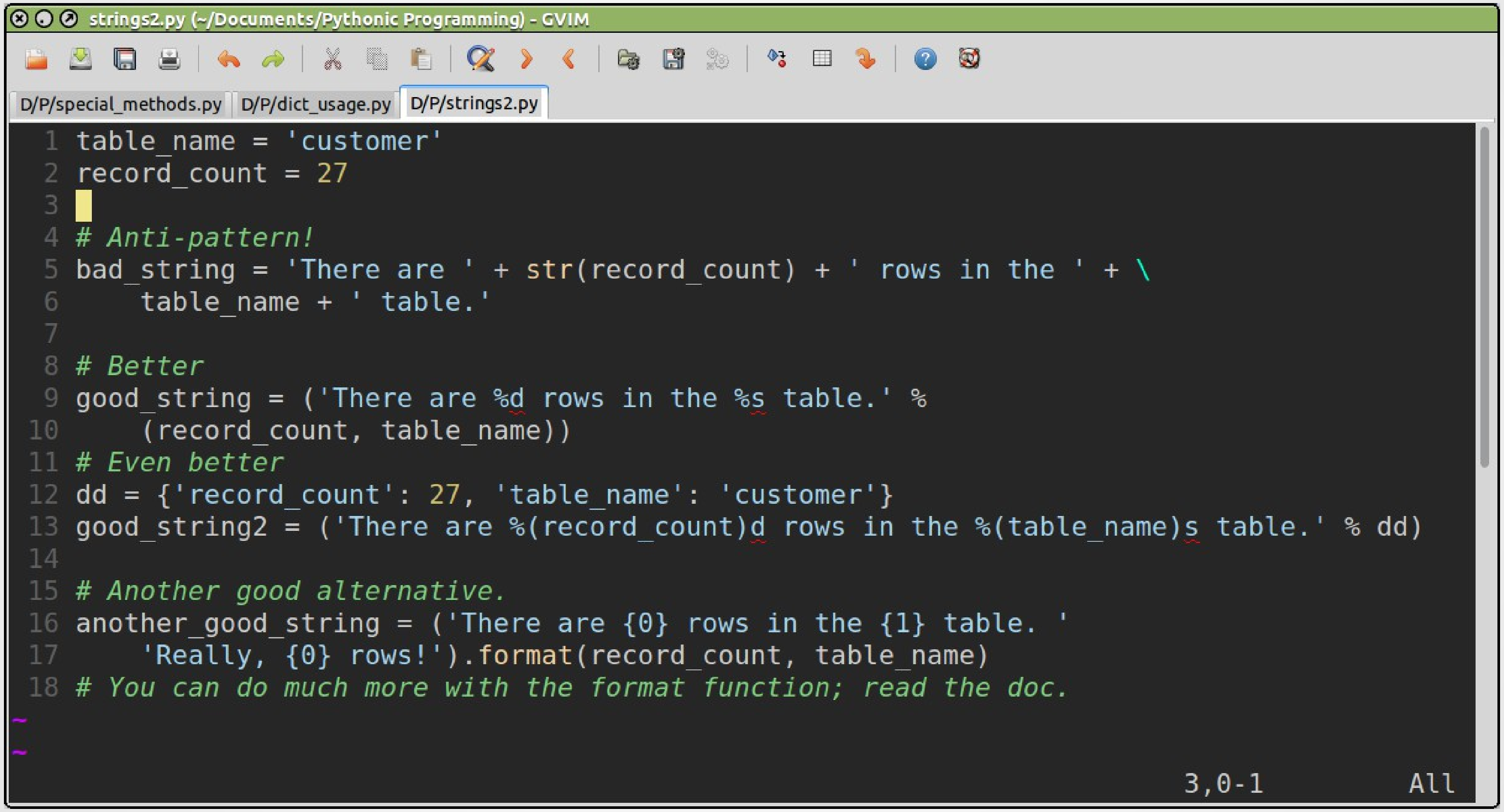
Good string habits


strings.py (~/Documents/Pythonic Programming) - GVIM

D/P/strings.py

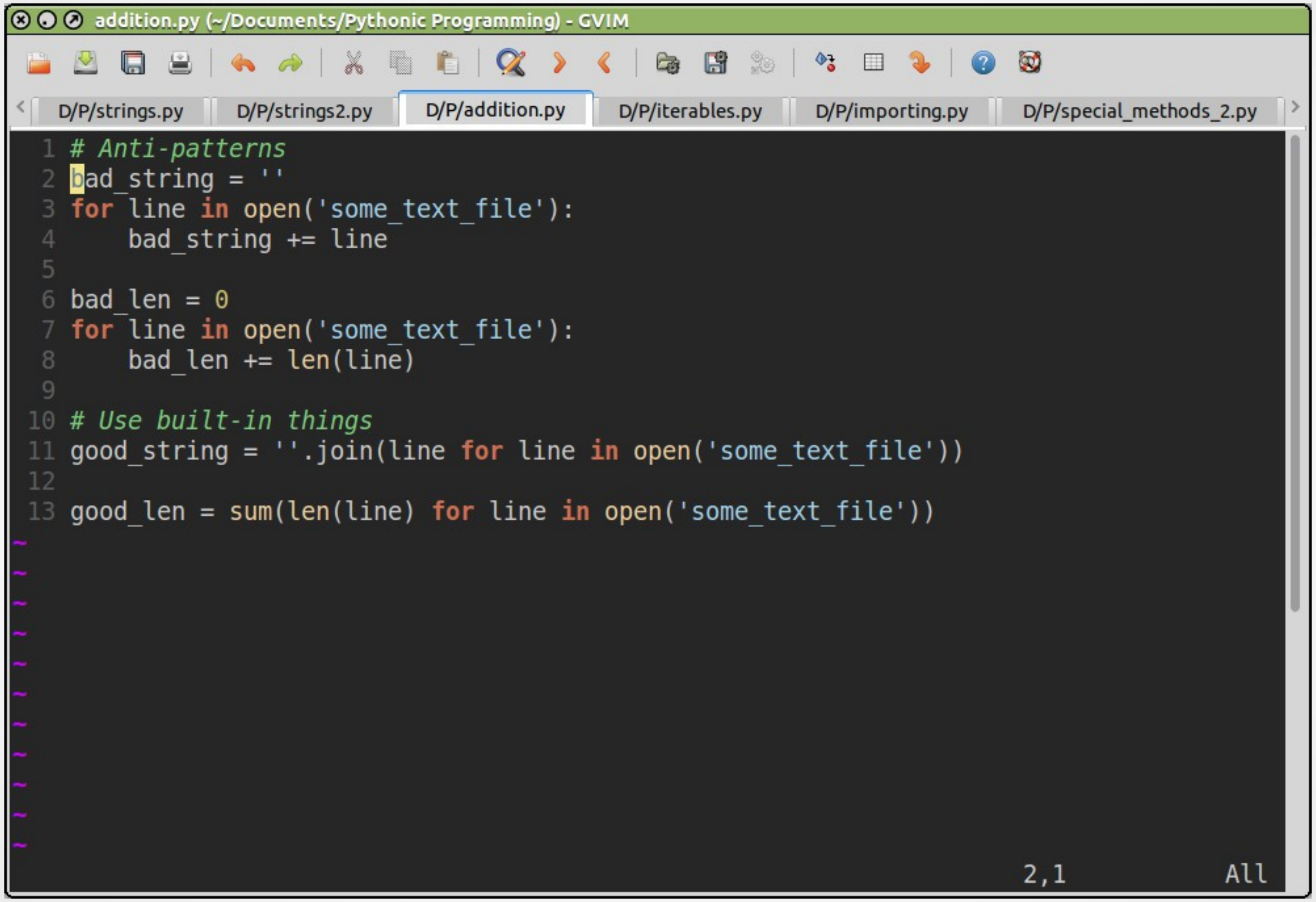
```
# Building strings like this will feed the garbage collector.
bad_string = 'This is the ' + \
    'wrong way to build ' + \
    'long strings in Python.'

# One good alternative
good_string = '''
You can build long strings like this,
but it may not match your indentation.
Also, your string will contain line breaks.
'''

# Another good alternative
good_string2 = (
    'You can build long strings '
    'and still keep your source '
    'tidy by putting short strings '
    'inside parentheses.  No line '
    'breaks will be added.'
    )
~
~
~
                                                  5,0-1         All
```

More strings



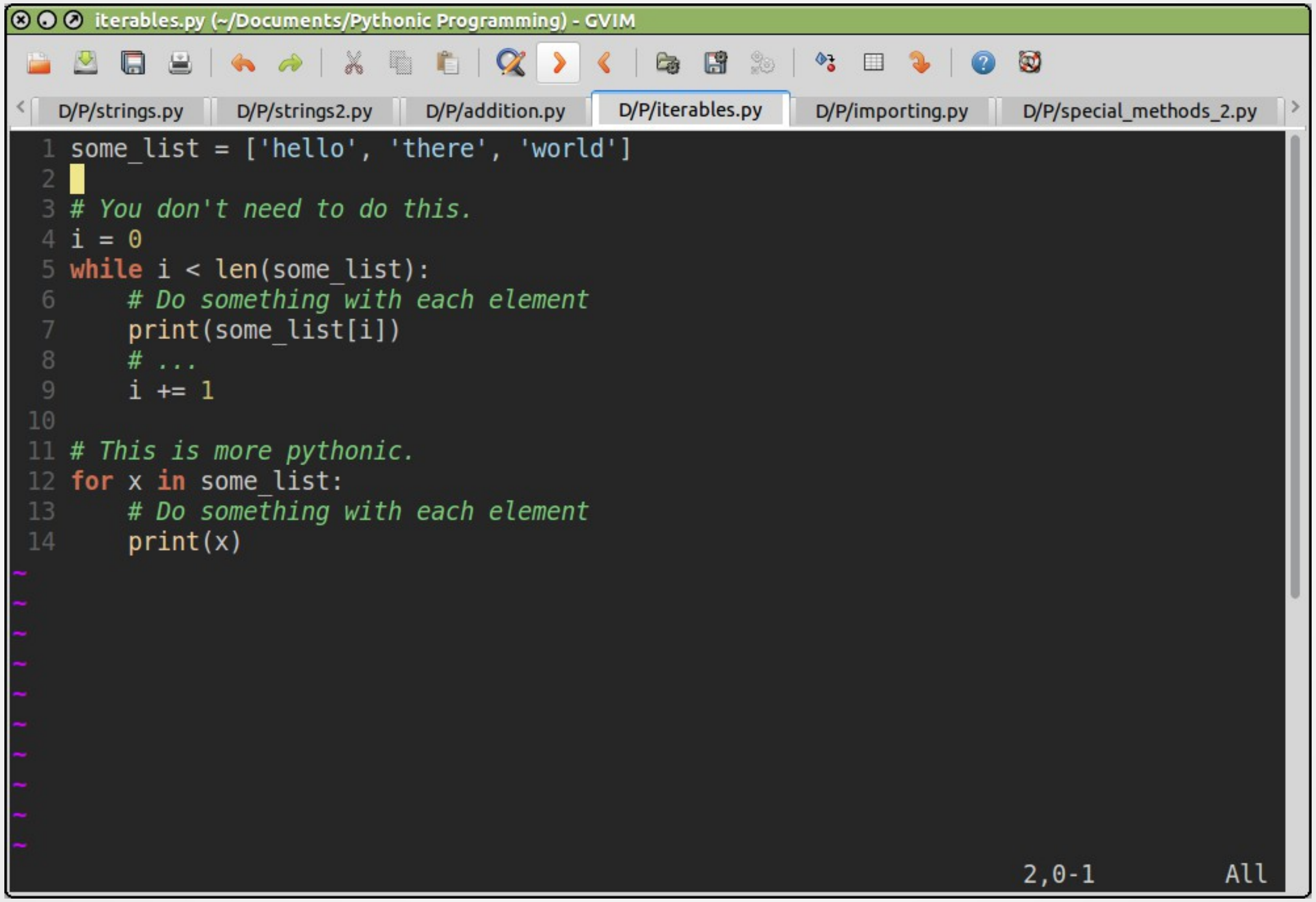
```python
table_name = 'customer'
record_count = 27

# Anti-pattern!
bad_string = 'There are ' + str(record_count) + ' rows in the ' + \
    table_name + ' table.'

# Better
good_string = ('There are %d rows in the %s table.' %
    (record_count, table_name))
# Even better
dd = {'record_count': 27, 'table_name': 'customer'}
good_string2 = ('There are %(record_count)d rows in the %(table_name)s table.' % dd)

# Another good alternative.
another_good_string = ('There are {0} rows in the {1} table. '
    'Really, {0} rows!').format(record_count, table_name)
# You can do much more with the format function; read the doc.
```

Use built-in functions



```python
# Anti-patterns
bad_string = ''
for line in open('some_text_file'):
    bad_string += line

bad_len = 0
for line in open('some_text_file'):
    bad_len += len(line)

# Use built-in things
good_string = ''.join(line for line in open('some_text_file'))

good_len = sum(len(line) for line in open('some_text_file'))
```

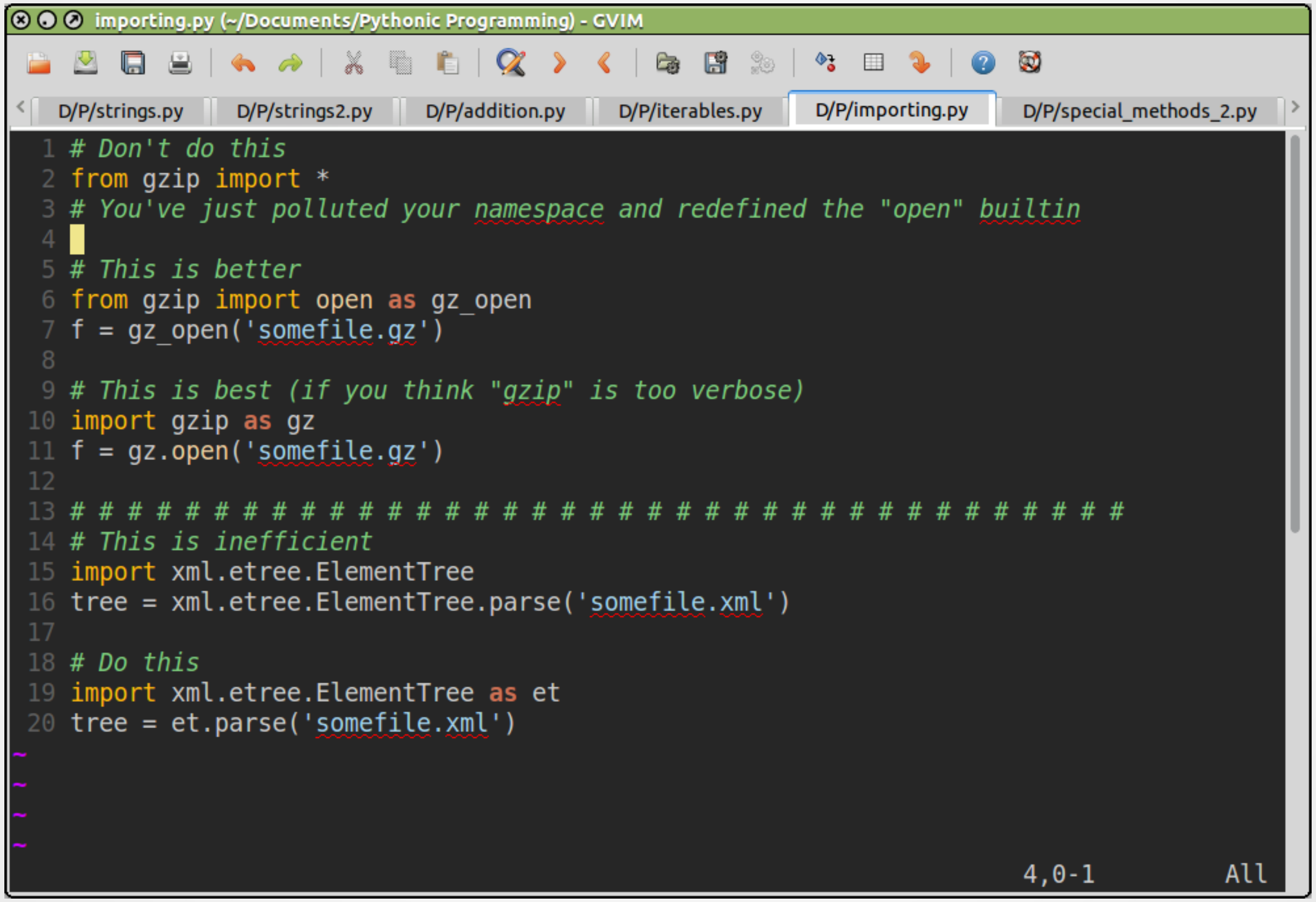
Avoid indexing into lists



iterables.py (~/Documents/Pythonic Programming) - GVIM

```python
some_list = ['hello', 'there', 'world']

# You don't need to do this.
i = 0
while i < len(some_list):
    # Do something with each element
    print(some_list[i])
    # ...
    i += 1

# This is more pythonic.
for x in some_list:
    # Do something with each element
    print(x)
```

Import carefully

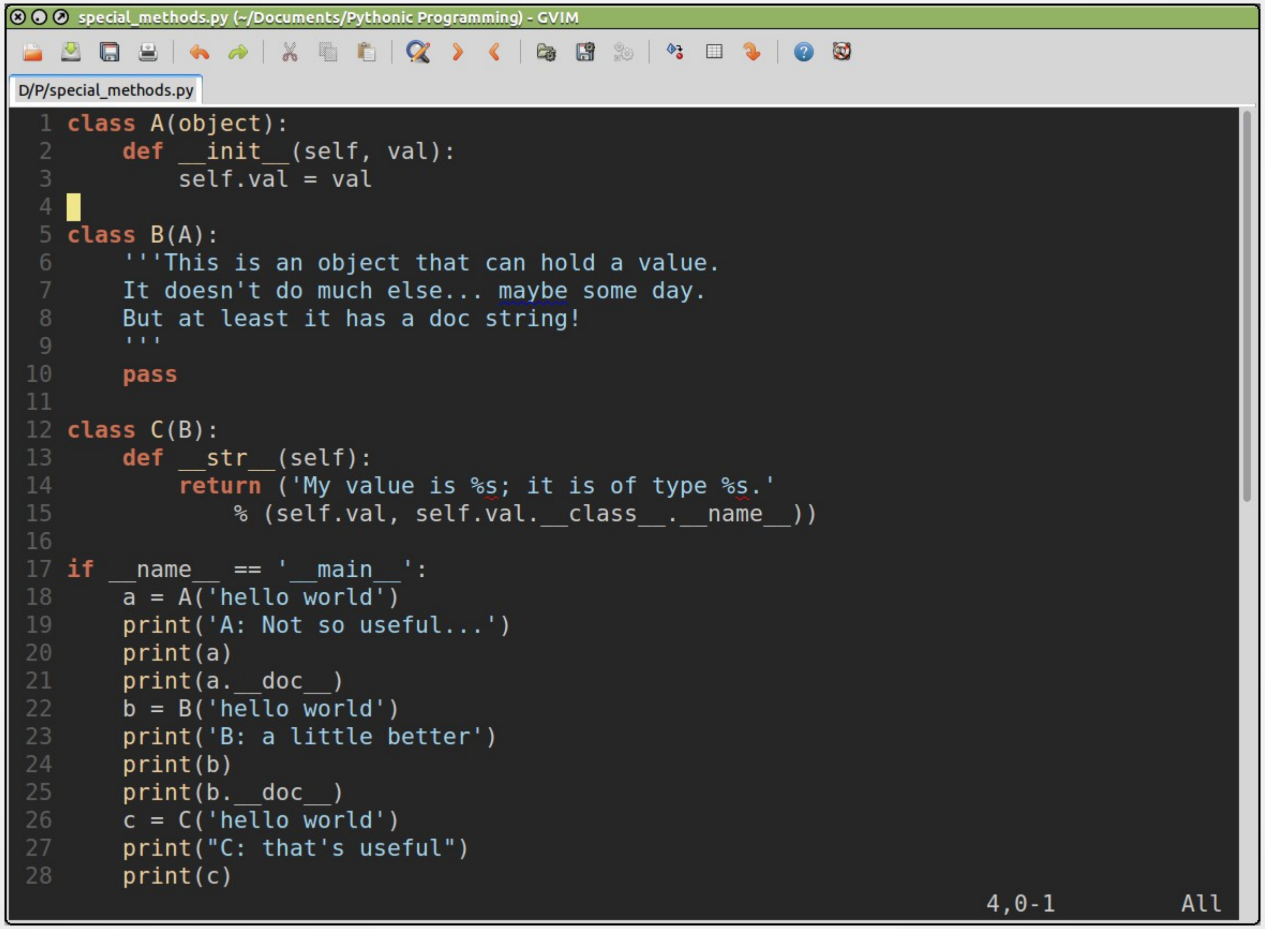


```python
# Don't do this
from gzip import *
# You've just polluted your namespace and redefined the "open" builtin

# This is better
from gzip import open as gz_open
f = gz_open('somefile.gz')

# This is best (if you think "gzip" is too verbose)
import gzip as gz
f = gz.open('somefile.gz')

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# This is inefficient
import xml.etree.ElementTree
tree = xml.etree.ElementTree.parse('somefile.xml')

# Do this
import xml.etree.ElementTree as et
tree = et.parse('somefile.xml')
~
~
~
~
                                                      4,0-1         All
```

Patterns

- Python has some good things that you should take advantage of

Take advantage of special methods



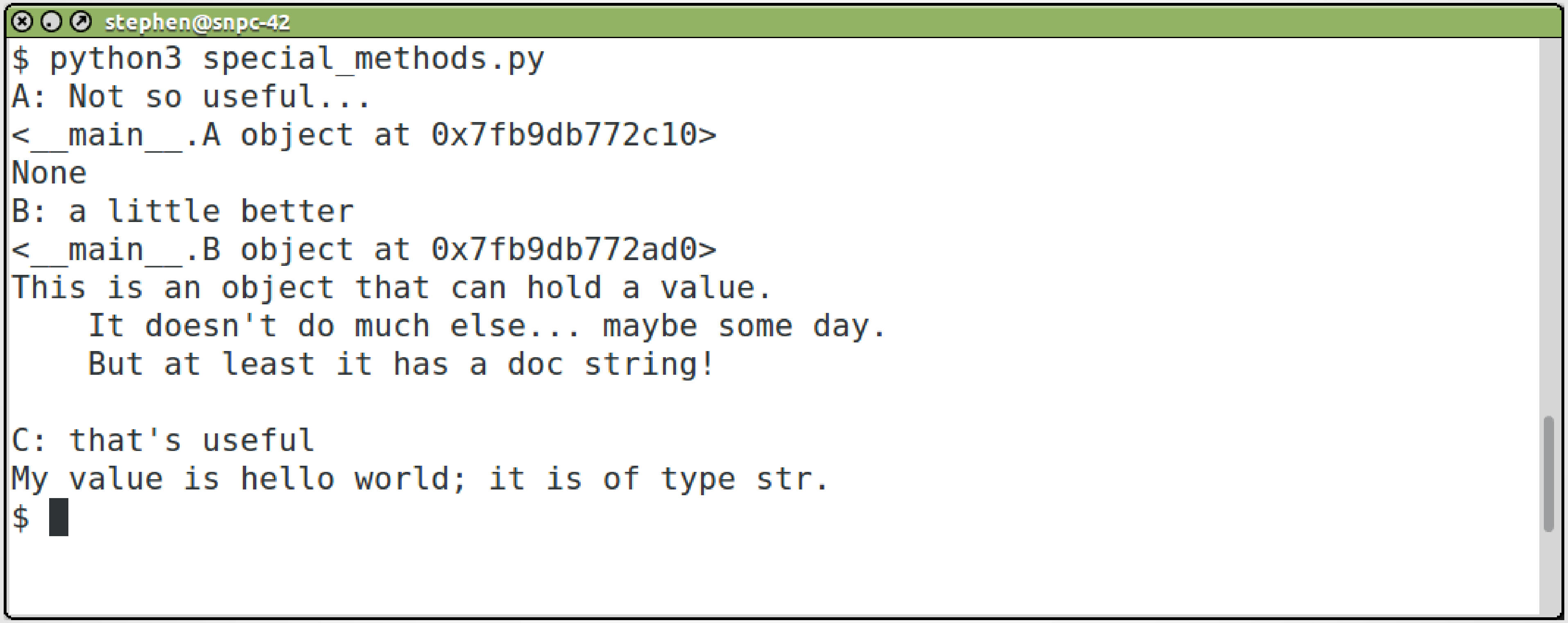
special_methods.py (~/Documents/Pythonic Programming) - GVIM

D/P/special_methods.py

```python
class A(object):
    def __init__(self, val):
        self.val = val

class B(A):
    '''This is an object that can hold a value.
    It doesn't do much else... maybe some day.
    But at least it has a doc string!
    '''
    pass

class C(B):
    def __str__(self):
        return ('My value is %s; it is of type %s.'
            % (self.val, self.val.__class__.__name__))

if __name__ == '__main__':
    a = A('hello world')
    print('A: Not so useful...')
    print(a)
    print(a.__doc__)
    b = B('hello world')
    print('B: a little better')
    print(b)
    print(b.__doc__)
    c = C('hello world')
    print("C: that's useful")
    print(c)
```

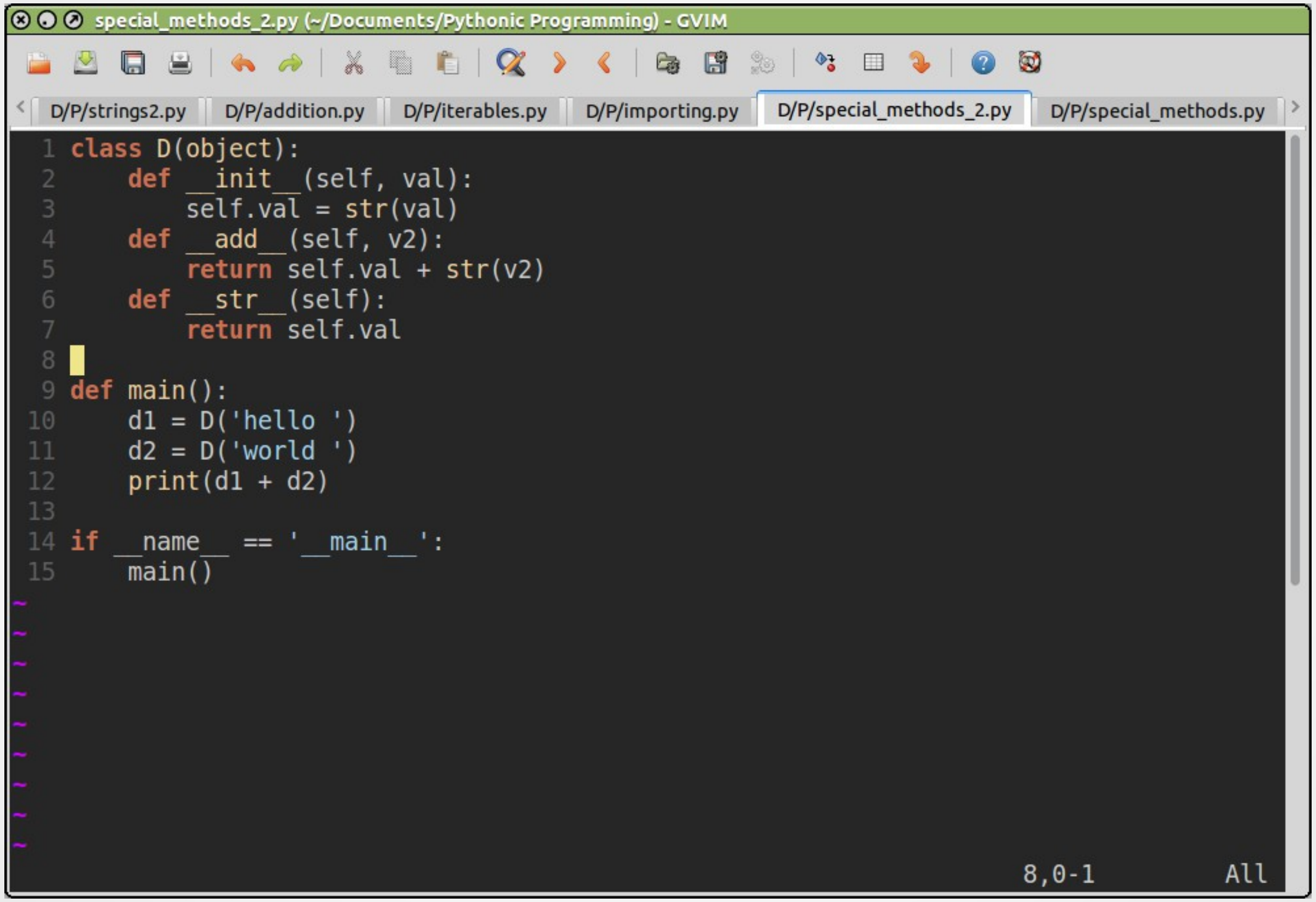
```
4,0-1          All
```

...and the output is


stephen@snpc-42

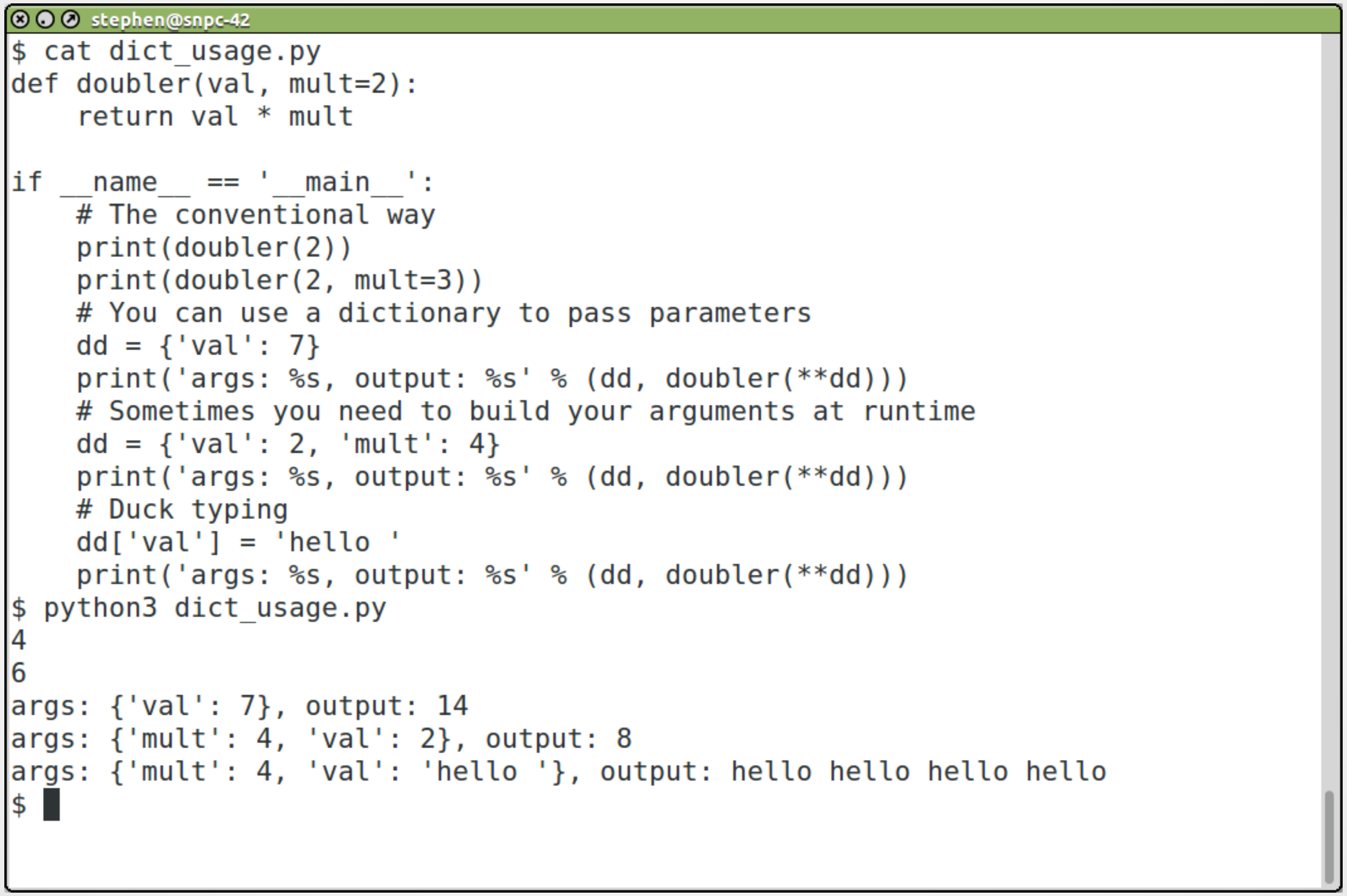
```
$ python3 special_methods.py
A: Not so useful...
<__main__.A object at 0x7fb9db772c10>
None
B: a little better
<__main__.B object at 0x7fb9db772ad0>
This is an object that can hold a value.
    It doesn't do much else... maybe some day.
    But at least it has a doc string!

C: that's useful
My value is hello world; it is of type str.
$ 
```

Use special methods to make your objects work with Python operators



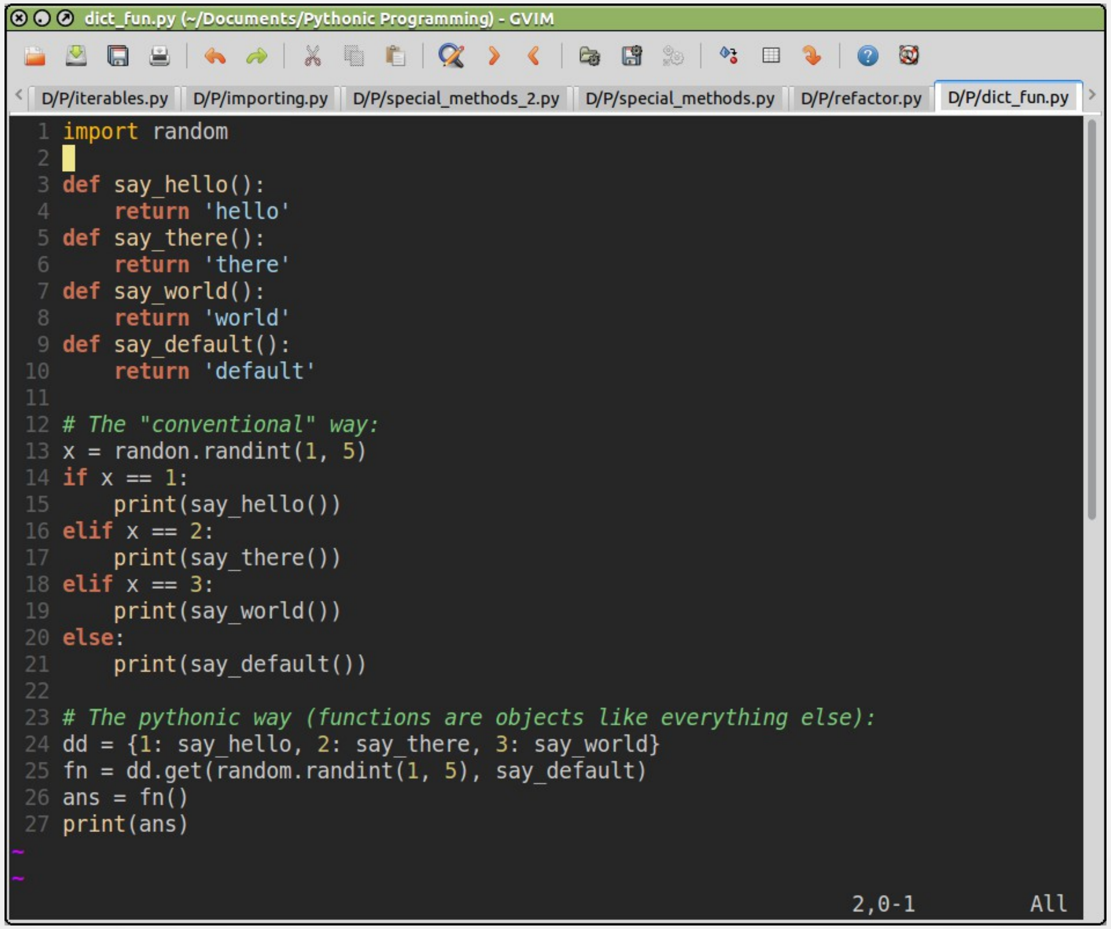
```python
class D(object):
    def __init__(self, val):
        self.val = str(val)
    def __add__(self, v2):
        return self.val + str(v2)
    def __str__(self):
        return self.val

def main():
    d1 = D('hello ')
    d2 = D('world ')
    print(d1 + d2)

if __name__ == '__main__':
    main()
```

Pass arguments with dictionaries



```
$ cat dict_usage.py
def doubler(val, mult=2):
    return val * mult

if __name__ == '__main__':
    # The conventional way
    print(doubler(2))
    print(doubler(2, mult=3))
    # You can use a dictionary to pass parameters
    dd = {'val': 7}
    print('args: %s, output: %s' % (dd, doubler(**dd)))
    # Sometimes you need to build your arguments at runtime
    dd = {'val': 2, 'mult': 4}
    print('args: %s, output: %s' % (dd, doubler(**dd)))
    # Duck typing
    dd['val'] = 'hello '
    print('args: %s, output: %s' % (dd, doubler(**dd)))
$ python3 dict_usage.py
4
6
args: {'val': 7}, output: 14
args: {'mult': 4, 'val': 2}, output: 8
args: {'mult': 4, 'val': 'hello '}, output: hello hello hello hello
$
```

More fun with dictionaries



dict_fun.py (~/Documents/Pythonic Programming) - GVIM

D/P/iterables.py | D/P/importing.py | D/P/special_methods_2.py | D/P/special_methods.py | D/P/refactor.py | D/P/dict_fun.py

```python
import random

def say_hello():
    return 'hello'
def say_there():
    return 'there'
def say_world():
    return 'world'
def say_default():
    return 'default'

# The "conventional" way:
x = randon.randint(1, 5)
if x == 1:
    print(say_hello())
elif x == 2:
    print(say_there())
elif x == 3:
    print(say_world())
else:
    print(say_default())

# The pythonic way (functions are objects like everything else):
dd = {1: say_hello, 2: say_there, 3: say_world}
fn = dd.get(random.randint(1, 5), say_default)
ans = fn()
print(ans)
```

2,0-1 All

Summary

- Python is different
- Understanding how it works will help you write more pythonic code
- Know its strengths and limitations

Resources

- PEP 8 http://www.python.org/dev/peps/pep-0008/
- Python reference http://docs.python.org/3.3/reference/datamodel.html
- Python cookbook http://code.activestate.com/recipes/langs/python/
- Stack overflow
- Python package index https://pypi.python.org/pypi
- Cookie Monster http://www.youtube.com/watch?v=ul9MtMiIOnE