Python Gotchas

Adrián Matellanes

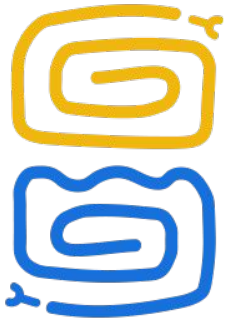
Lead API Developer at **@EburyES** & **Málaga Python Organizer**

twitter.com/_amatellanes

github.com/amatellanes

We’re hiring!

https://careers.ebury.com/



MÁLAGA
PYTHON

Join us!

https://www.meetup.com/malaga-python/

La Térmica - Málaga
Saturday, 6 de May, 2017

Call For Papers

www.opensouthcode.org

What is a “gotcha”?

A gotcha is a valid construct in a programming language that works as documented but is counter-intuitive and almost invites mistakes.

Learning the Zen of Python

Learning the Zen of Python

```
>>> import this
```

https://www.python.org/dev/peps/**pep-0020**/

Learning the Zen of Python

```
>>> import this
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

https://www.python.org/dev/peps/**pep-0020**/

Division Operator

Division Operator

```
>>> 5 / 2  # Python 2
```

```
>>> 5 // 2  # Python 2
```

```
>>> 5.0 / 2  # Python 2
```

```
>>> 5.0 // 2  # Python 2
```

Division Operator

```
>>> 5 / 2  # Python 2
2

>>> 5 // 2  # Python 2
2

>>> 5.0 / 2  # Python 2
2.5

>>> 5.0 // 2  # Python 2
2.0
```

Division Operator

```
>>> 5 / 2  # Python 3
```

```
>>> 5 // 2  # Python 3
```

```
>>> 5.0 / 2  # Python 3
```

```
>>> 5.0 // 2  # Python 3
```

Division Operator

```
>>> 5 / 2  # Python 3
2.5

>>> 5 // 2  # Python 3
2

>>> 5.0 / 2  # Python 3
2.5

>>> 5.0 // 2  # Python 3
2.0
```

Division Operator

```
>>> from __future__ import division
>>> 5 / 2  # Python 2 or 3
2.5
```

https://www.python.org/dev/peps/**pep-0238**/

Maths?

Maths?

```
>>> True + 3
```

```
>>> False * 4
```

```
>>> True / False
```

Maths?

```
>>> True + 3
4

>>> False * 4
0

>>> True / False
ZeroDivisionError: integer division or modulo by zero
```

Maths?

```
>>> 'a' * 5
```

```
>>> 'b' + 4
```

Maths?

```
>>> 'a' * 5
'aaaaa'

>>> 'b' + 4
TypeError: cannot concatenate 'str' and 'int' objects
```

Forgetting parentheses

Forgetting parentheses

```
>>> def foo():
...     return None
...
>>> foo is None


>>> foo() is None
```

Forgetting parentheses

```
>>> def foo():
...     return None
...
>>> foo is None
False

>>> foo() is None
True
```

Chaining comparisons

Chaining comparisons

```
>>> (42 > 23) == True
```

```
>>> 42 > (23 == True)
```

```
>>> 42 > 23 == True
```

Chaining comparisons

```
>>> (42 > 23) == True
True

>>> 42 > (23 == True)
True

>>> 42 > 23 == True
False
```

https://docs.python.org/3/reference/expressions.html#**operator-precedence**

Chaining comparisons

```
>>> 42 > 23 == True
False

>>> 42 > 23 and 23 == True
False
```

https://docs.python.org/3/reference/expressions.html#**comparisons**

`is not` **Identity Operator**

`is not` Identity Operator

```
>>> 2 is not None
```

```
>>> 2 is (not None)
```

`is not` Identity Operator

```
>>> 2 is not None
True

>>> 2 is (not None)
False
```

Lazy Binding

Lazy Binding

```
>>> x = 23
>>> f = lambda: x
>>> x = 42
>>> print(f())
```

Lazy Binding

```
>>> x = 23
>>> f = lambda: x
>>> x = 42
>>> print(f())
42
```

Lazy Binding

```
>>> x = 23
>>> f = lambda x=x: x
>>> x = 42
>>> print(f())
23
```

Lazy Binding

```
>>> funcs = []
>>> for i in range(10):
...     funcs.append(lambda: i ** 2)
...
>>> print([f() for f in funcs])
```

Lazy Binding

```
>>> funcs = []
>>> for i in range(10):
...     funcs.append(lambda: i ** 2)
...
>>> print([f() for f in funcs])
[81, 81, 81, 81, 81, 81, 81, 81, 81, 81]
```

Lazy Binding

```
>>> funcs = []
>>> for i in range(10):
...     funcs.append(lambda i=i: i ** 2)
...
>>> print([f() for f in funcs])
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Evaluation Time Discrepancy

Evaluation Time Discrepancy

```
>>> seq = [4, 8, 15, 16, 16, 23, 42]
>>> g = (x for x in seq if x in seq)
>>> seq = [16]
>>> print(list(g))
```

https://www.python.org/dev/peps/**pep-0289**/

Evaluation Time Discrepancy

```
>>> seq = [4, 8, 15, 16, 16, 23, 42]
>>> g = (x for x in seq if x in seq)
>>> seq = [16]
>>> print(list(g))
[16, 16]
```

https://www.python.org/dev/peps/**pep-0289**/

Memory Management

Memory Management

```
>>> list1 = [1, 2, 3]
>>> list2 = list1
>>> list1[1] = 5
>>> print(list1)


>>> print(list2)
```

Memory Management

```
>>> list1 = [1, 2, 3]
>>> list2 = list1
>>> list1[1] = 0
>>> print(list1)
[1, 0, 3]

>>> print(list2)
[1, 0, 3]
```

Memory Management

```
>>> list1 = [1, 2, 3]
>>> list2 = list1[:]
>>> list1[1] = 0
>>> print(list1)
[1, 0, 3]

>>> print(list2)
[1, 2, 3]
```

Memory Management

```
>>> from copy import copy
>>> list1 = [1, 2, 3]
>>> list2 = copy(list1)
>>> list1[1] = 5
>>> print(list1)
[1, 0, 3]

>>> print(list2)
[1, 2, 3]
```

Memory Management

```
>>> list1 = [{'a': [1, 2, 3]}, {'b': [4, 5, 6]}]
>>> list2 = list1
>>> list1[0]['a'][1] = 0
>>> print(list1)


>>> print(list2)
```

Memory Management

```
>>> list1 = [{'a': [1, 2, 3]}, {'b': [4, 5, 6]}]
>>> list2 = list1
>>> list1[0]['a'][1] = 0
>>> print(list1)
[{'a': [1, 0, 3]}, {'b': [4, 5, 6]}]

>>> print(list2)
[{'a': [1, 0, 3]}, {'b': [4, 5, 6]}]
```

Memory Management

```
>>> from copy import copy
>>> list1 = [{'a': [1, 2, 3]}, {'b': [4, 5, 6]}]
>>> list2 = copy(list1)
>>> list1[0]['a'][1] = 0
>>> print(list1)
[{'a': [1, 0, 3]}, {'b': [4, 5, 6]}]

>>> print(list2)
[{'a': [1, 0, 3]}, {'b': [4, 5, 6]}]
```

Memory Management

```
>>> from copy import deepcopy
>>> list1 = [{'a': [1, 2, 3]}, {'b': [4, 5, 6]}]
>>> list2 = deepcopy(list1)
>>> list1[0]['a'][1] = 0
>>> print(list1)
[{'a': [1, 0, 3]}, {'b': [4, 5, 6]}]

>>> print(list2)
[{'a': [1, 2, 3]}, {'b': [4, 5, 6]}]
```

Default Mutable Argument

Default Mutable Argument

```
>>> def foo(x=[]):
...     x.append(1)
...     print(x)
...
>>> foo()


>>> foo()
```

Default Mutable Argument

```
>>> def foo(x=[]):
...     x.append(1)
...     print(x)
...
>>> foo()
[1]

>>> foo()
[1, 1]
```

Default Mutable Argument

```
>>> def foo(x=None):
...     if x is None:
...         x = []
...     x.append(1)
...     print(x)
...
>>> foo()
[1]

>>> foo()
[1]
```

Default Mutable Argument

```
>>> def foo(x=None):
...     x = x or []
...     x.append(1)
...     print(x)
...
>>> foo()
[1]

>>> foo()
[1]
```

Unpacking

Unpacking

```
>>> a, b, c = 1, 2, 3
>>> a


>>> b


>>> c
```

Unpacking

```
>>> a, b, c = 1, 2, 3
>>> a
1

>>> b
2

>>> c
3
```

Unpacking

```
>>> a, b, c = [1, 2, 3]
>>> a


>>> b


>>> c
```

Unpacking

```
>>> a, b, c = [1, 2, 3]
>>> a
1

>>> b
2

>>> c
3
```

Unpacking

```
>>> a, b, c = (2 * i + 1 for i in range(3))
>>> a


>>> b


>>> c
```

Unpacking

```
>>> a, b, c = (2 * i + 1 for i in range(3))
>>> a
1

>>> b
2

>>> c
3
```

Unpacking

```
>>> a, b, c = [1, (2, 3), 4]
>>> a


>>> b


>>> c
```

Unpacking

```
>>> a, b, c = [1, (2, 3), 4]
>>> a
1

>>> b
(2, 3)

>>> c
3
```

Unpacking

```
>>> a, b, c = [1, 2, 3, 4]  # Python 2
>>> a


>>> b


>>> c
```

Unpacking

```
>>> a, b, c = [1, 2, 3, 4]  # Python 2
ValueError: too many values to unpack
```

Unpacking

```
>>> a, b, c = [1, 2, 3, 4]  # Python 3
>>> a

>>> b


>>> c
```

Unpacking

```
>>> a, b, c = [1, 2, 3, 4]  # Python 3
ValueError: too many values to unpack (expected 3)
```

Unpacking

```
>>> a, *b, c = [1, 2, 3, 4]  # Only Python 3
>>> a


>>> b


>>> c
```

Unpacking

```
>>> a, *b, c = [1, 2, 3, 4]  # Only Python 3
>>> a
1

>>> b
[2, 3]

>>> c
4
```

Slicing list

Slicing list

```
>>> a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> a[::2]


>>> a[::3]


>>> a[2:8:2]
```

Slicing list

```
>>> a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> a[::2]
[0, 2, 4, 6, 8, 10]

>>> a[::3]
[0, 3, 6, 9]

>>> a[2:8:2]
[2, 4, 6]
```

Dictionary and Set Comprehensions

Dictionary and Set Comprehensions

```
>>> {x: x ** 2 for x in range(5)}
```

```
>>> {x for x in range(5)}
```

Dictionary and Set Comprehensions

```
>>> {x: x ** 2 for x in range(5)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

>>> {x for x in range(5)}
{0, 1, 2, 3, 4}
```

Dictionary and Set Comprehensions

```
>>> type({})
```

Dictionary and Set Comprehensions

```
>>> type({})
dict
```

Bytecode Files Everywhere!

Bytecode Files Everywhere!

Disabling bytecode (.pyc) files

```
$ export PYTHONDONTWRITEBYTECODE=1
```

https://www.python.org/dev/peps/**pep-0304**/

Bytecode Files Everywhere!

Removing bytecode (.pyc) files

```
$ find . -type f -name "*.py[co]" -delete -or \
           -type d -name "__pycache__" -delete
$ pyclean .
$ py3clean .
```

http://manpages.ubuntu.com/manpages/xenial/man1/pyclean.1.html
http://manpages.ubuntu.com/manpages/xenial/man1/py3clean.1.html

Using class variables incorrectly

Using class variables incorrectly

```
>>> class A(object):
...     x = 1
>>> class B(A):
...     pass
>>> class C(A):
...     pass
>>> print(A.x, B.x, C.x)
```

Using class variables incorrectly

```
>>> class A(object):
...     x = 1
>>> class B(A):
...     pass
>>> class C(A):
...     pass
>>> print(A.x, B.x, C.x)
1 1 1
```

Using class variables incorrectly

```
>>> class A(object):
...     x = 1
>>> class B(A):
...     pass
>>> class C(A):
...     pass
>>> B.x = 2
>>> A.x = 3
>>> print(A.x, B.x, C.x)
```

Using class variables incorrectly

```
>>> class A(object):
...     x = 1
>>> class B(A):
...     pass
>>> class C(A):
...     pass
>>> B.x = 2
>>> A.x = 3
>>> print(A.x, B.x, C.x)
3 2 3
```

Catching multiple exceptions

Catching multiple exceptions

```
>>> try:
...     x = int('a')
... except ValueError, IndexError:  # In Python 2
...     print 'Error'
```

Catching multiple exceptions

```
>>> try:
...     x = int('a')
... except ValueError, IndexError:  # In Python 2
...     print 'Error'
'Error'
```

Catching multiple exceptions

```
>>> try:
...     x = int('a')
... except ValueError, IndexError:  # In Python 3
...     print('Error')
SyntaxError: invalid syntax
```

Catching multiple exceptions

```
>>> try:
...     # do something that may fail
... except (ValueError, IndexError) as e:  # In Python 2 and 3
...     # do this if ANYTHING goes wrong
```

Misunderstanding scope rules

Misunderstanding scope rules

```
>>> x = 10
>>> def foo():
...     x += 1
...     print(x)
...
>>> foo()
```

Misunderstanding scope rules

```
>>> x = 10
>>> def foo():
...     x += 1
...     print(x)
...
>>> foo()
UnboundLocalError: local variable 'x' referenced before
assignment
```

Misunderstanding scope rules

```
>>> l = [1, 2, 3]
>>> def foo():
...     l.append(5)
...
>>> foo()
>>> l
```

Misunderstanding scope rules

```
>>> l = [1, 2, 3]
>>> def foo():
...     l.append(5)
...
>>> foo()
>>> l
[1, 2, 3, 5]
```

Misunderstanding scope rules

```
>>> l = [1, 2, 3]
>>> def foo():
...     l += [5]
...
>>> foo()
>>> l
```

Misunderstanding scope rules

```
>>> l = [1, 2, 3]
>>> def foo():
...     l += [5]
...
>>> foo()
>>> l
UnboundLocalError: local variable 'l' referenced before
assignment
```

Modifying a list while iterating over it

Modifying a list while iterating over it

```
>>> odd = lambda x : bool(x % 2)
>>> numbers = [n for n in range(10)]
>>> for i in range(len(numbers)):
...     if odd(numbers[i]):
...         del numbers[i]
...
```

Modifying a list while iterating over it

```
>>> odd = lambda x : bool(x % 2)
>>> numbers = [n for n in range(10)]
>>> for i in range(len(numbers)):
...     if odd(numbers[i]):
...         del numbers[i]
...
IndexError: list index out of range
```

Modifying a list while iterating over it

```
>>> odd = lambda x : bool(x % 2)
>>> numbers = [n for n in range(10)]
>>> numbers[:] = [n for n in numbers if odd(n)]
>>> numbers
[1, 3, 5, 7, 9]
```

Name clashing with modules and keywords

Name clashing with modules and keywords

```
>>> list = [1, 2]
>>> list()
```

Name clashing with modules and keywords

```
>>> list = [1, 2]
>>> list()
TypeError: 'list' object is not callable
```

Name clashing with modules and keywords

```
# email.py
a = 1
```

Name clashing with modules and keywords

```
>>> import email
>>> email.parser()
AttributeError: 'module' object has no attribute 'parser'
```

Obscuring import statements

Obscuring import statements

```
# module1.py
x = 1

# module2.py
x = 2

# module3.py
x = 3
```

Obscuring import statements

```
>>> from module1 import *
>>> from module2 import *
>>> from module3 import *
>>> print(x)
```

Obscuring import statements

```
>>> from module1 import *
>>> from module2 import *
>>> from module3 import *
>>> print(x)
3
```

Creating Circular Module Dependencies

Creating Circular Module Dependencies

```
# recur1.py
x = 1
import recur2
y = 2
```

Creating Circular Module Dependencies

```
# recur2.py
from recur1 import x
from recur1 import y
```

Creating Circular Module Dependencies

```
>>> import recur1
```

Creating Circular Module Dependencies

```
>>> import recur1
...
  File "recur1.py", line 2, in <module>
    import recur2
  File "recur2.py", line 2, in <module>
    from recur1 import y
ImportError: cannot import name y
```

Creating Circular Module Dependencies

```
# recur1.py
from recur2 import x
def f():
    return x
print(f())
```

Creating Circular Module Dependencies

```
# recur2.py
import recur1
x = 1
def g():
    print(recur1.f())
```

Creating Circular Module Dependencies

```
>>> import recur1
```

Creating Circular Module Dependencies

```
>>> import recur1
1
```

Creating Circular Module Dependencies

```
>>> import recur2
```

Creating Circular Module Dependencies

```
>>> import recur2
...
  File "/home/miusuario/tes/recur2.py", line 1, in <module>
    import recur1
  File "/home/miusuario/tes/recur1.py", line 1, in <module>
    from recur2 import x
ImportError: cannot import name 'x'
```

Creating Circular Module Dependencies

```
# recur2.py
x = 1
import recur1
def g():
    print(recur1.f())
```

Creating Circular Module Dependencies

```
>>> import recur2
1
```

Creating Circular Module Dependencies

```
# recur2.py
x = 1
def g():
    import recur1
    print(recur1.f())
```

Thank you!

Questions?

Slides available at
https://speakerdeck.com/amatellanes/python-gotchas