Common programming mistakes

Dariusz Śmigiel

PyCon PL 2014: 18.10.2014


XPS
DELL

Based on Martin Chikilian:

`www.toptal.com/python/top-10-mistakes-that-python-programmers-make`

Question?!

Let any one of you who is without sin be the first to throw a stone at her.

Function arguments

```
>>> def foo(bar=[]):
...     bar.append("baz")
...     return bar
```

Function arguments

```
>>> def foo(bar=[]):
...     bar.append("baz")
...     return bar

>>> foo()
['baz']
```

```
>>> foo()
['baz', 'baz']
```

```
>>> foo()
['baz', 'baz']
```

```
>>> foo()
['baz', 'baz']
```


```
>>> foo()
['baz', 'baz', 'baz']
```

```
>>> foo()
['baz', 'baz']
```


```
>>> foo()
['baz', 'baz', 'baz']
```

Early binding

- initialization on function definition;
- by purpose: compiler associates identifier name (function or variable) with machine address

```
>>> def foo(bar=None):
...     if bar is None:
...         bar = []
...     bar.append("baz")
...     return bar
```

```
>>> def foo(bar=None):
...     if bar is None:
...         bar = []
...     bar.append("baz")
...     return bar

>>> foo()
['baz']
>>> foo()
['baz']
```

```
>>> def foo(bar=[]):
...     bar.append("baz")
...     return bar
```

```
3           0 LOAD_FAST                0 (bar)
            3 LOAD_ATTR                0 (append)
            6 LOAD_CONST               1 ('baz')
            9 CALL_FUNCTION            1
           12 POP_TOP

4          13 LOAD_FAST                0 (bar)
           16 RETURN_VALUE
```

```
>>> def foo(bar=None):
...     if bar is None:
...         bar = []
...     bar.append("baz")
...     return bar

  7           0 LOAD_FAST                0 (bar)
              3 POP_JUMP_IF_TRUE        15

  8           6 BUILD_LIST               0
              9 STORE_FAST               0 (bar)
             12 JUMP_FORWARD             0 (to 15)

  9     >>   15 LOAD_FAST                0 (bar)
             18 LOAD_ATTR                0 (append)
             21 LOAD_CONST               1 ('baz')
             24 CALL_FUNCTION            1
             27 POP_TOP

 10          28 LOAD_FAST                0 (bar)
             31 RETURN_VALUE
```

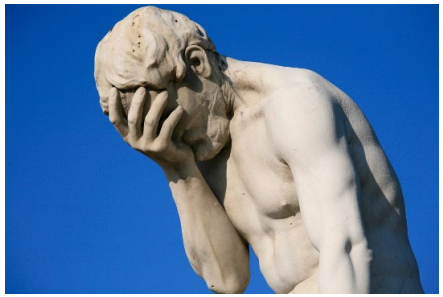
Binding variables in closures

```
>>> def create_multipliers():
...     return [lambda x : i * x for i in range(5)]
>>> for multiplier in create_multipliers():
...     print(multiplier(2))
...
```

Binding variables in closures

```
>>> def create_multipliers():
...     return [lambda x : i * x for i in range(5)]
>>> for multiplier in create_multipliers():
...     print(multiplier(2))
...
```

```
8
8
8
8
8
```



Late binding

- values of variables in closures are looked up at the time the inner function is called;
- by then, loop has completed and `i` returns 4

```
>>> def create_multipliers():
...     return [lambda x, i=i : i * x for i in
    ↪ range(5)]
...
>>> for multiplier in create_multipliers():
...     print (multiplier(2))
...
...
```

```
>>> def create_multipliers():
...     return [lambda x, i=i : i * x for i in
    ↪ range(5)]
...
>>> for multiplier in create_multipliers():
...     print (multiplier(2))
...
...

0
2
4
6
8
>>>
```

```
>>> def create_multipliers():
...     return [lambda x, i=i : i * x for i in
    ↪ range(5)]
...
>>> for multiplier in create_multipliers():
...     print (multiplier(2))
...
...

0
2
4
6
8
>>>
```

```
>>> def get_func(i):
...     return lambda x: i * x
...
>>> def create_multipliers():
...     return [get_func(i) for i in range(5)]
...
>>> for multiplier in create_multipliers():
...     print (multiplier(2))
...
...
0
2
4
6
8
>>>
```

Local names

```
x = 99
>>> def func():
...     print(x)
...
...
>>> func()
99
```

Local names

```
>>> x = 99
>>> def func():
...     print(x)
...     x = 88
...
```

Local names

```
>>> x = 99
>>> def func():
...     print(x)
...     x = 88
...

>>> func()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 2, in func
UnboundLocalError: local variable 'x' referenced
    ↪ before assignment
```

Local names

- Python sees the assignment to `x`
- it's decided that `x` is a local variable in function
- when function is run, assignment hasn't yet happened
- Python raises undefined error

```
>>> x = 99
>>> def func():
...     global x
...     print(x)
...     x = 88
...
```

```
>>> x = 99
>>> def func():
...     global x
...     print(x)
...     x = 88
...

>>> func()
99
```

Scope rules

```
>>> bar = 0
>>> def foo():
...     bar += 1
...     print(bar)
...
```

Scope rules

```
>>> bar = 0
>>> def foo():
...     bar += 1
...     print(bar)
...

>>> foo()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 2, in foo
UnboundLocalError: local variable 'bar' referenced
    ↪ before assignment
```

Python scope resolution is based on `LEGB`:

- Local - names assigned in any way within a function (`def` or `lambda`) and not declared global in that function;
- Enclosing function locals - name in the local scope of any and all enclosing (`def` or `lambda`), from inner to outer;
- Global (module) - names assigned at the top-level of a module file, or declared global in a `def` within the file;
- Built-in (Python) - names preassigned in a built-in names module: `open`, `range`, `SyntaxError`, ...

So, when you make an assignment to variable, Python considers it as in local scope. It shadows everything, that is outside this scope. In this case, we don't "see" variable `i` declared before function definition.

```
>>> bar = 0
>>> def foo(bar=bar):
...     bar += 1
...     return bar
...
```

```
>>> bar = 0
>>> def foo(bar=bar):
...     bar += 1
...     return bar
...

>>> foo()
1
>>> foo()
1
```

```
>>> bar = 0
>>> def foo(bar=bar):
...     bar += 1
...     return bar
...

>>> foo()
1
>>> foo()
1
```

```
31
32 bar = 0
33 def foo():
34     bar += 1
35     return bar
~
~
[34,1 Wszystko] examples.py  (unix)
local variable 'bar' (defined in enclosing scope on line 32) referenced before assignment
```

Class variables

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

Class variables

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

>>> print(A.x, B.x, C.x)
1 1 1
```

Class variables

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

>>> print(A.x, B.x, C.x)
1 1 1

>>> B.x = 2
```

Class variables

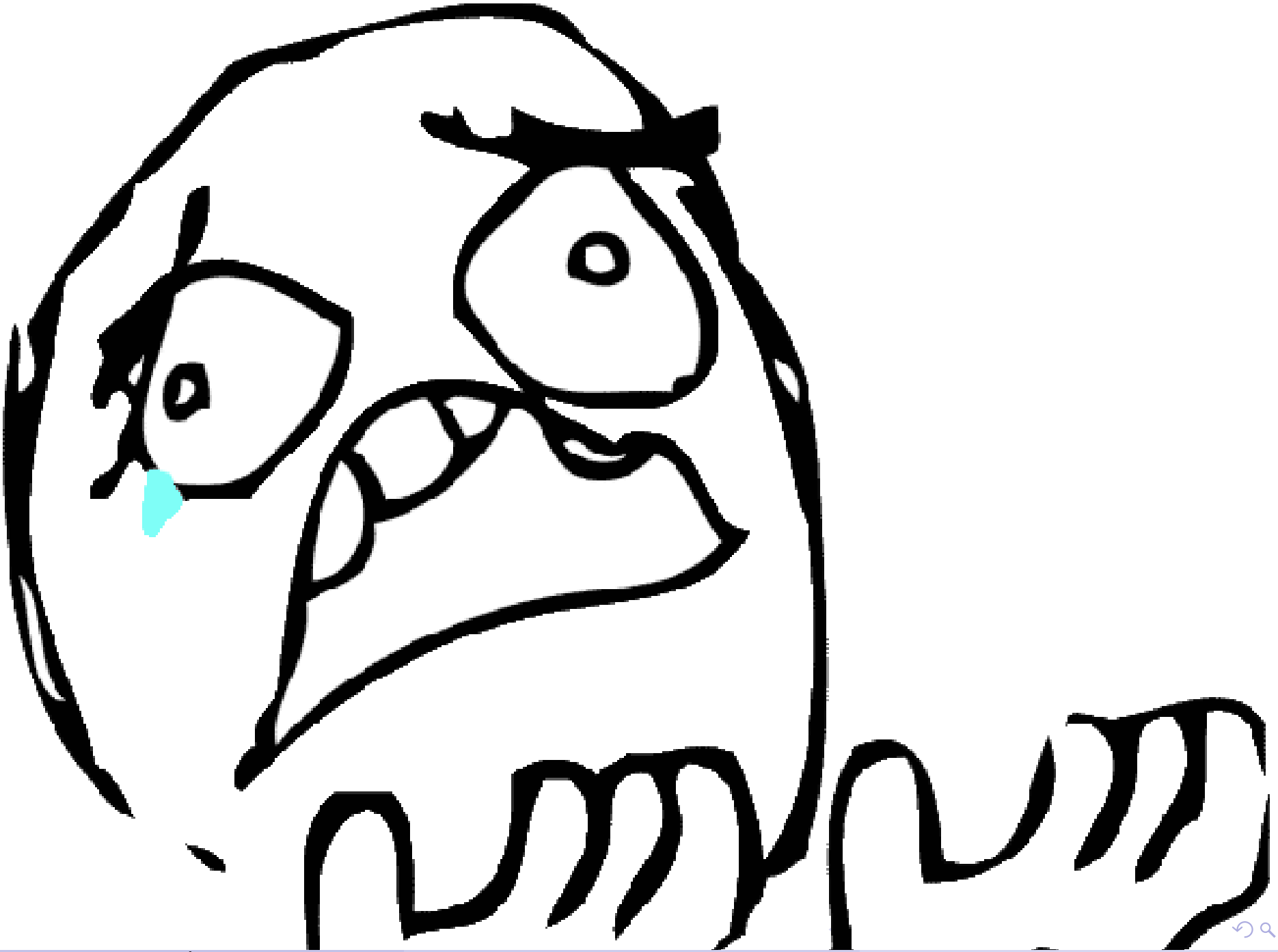
```
>>> class A(object):
...     x = 1
...
>>> class B(A):
...     pass
...
>>> class C(A):
...     pass

>>> print(A.x, B.x, C.x)
1 1 1

>>> B.x = 2

>>> print(A.x, B.x, C.x)
1 2 1
```

Class variables

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

>>> print(A.x, B.x, C.x)
1 1 1

>>> B.x = 2

>>> print(A.x, B.x, C.x)
1 2 1

>>> A.x = 3
```

Class variables

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

>>> print(A.x, B.x, C.x)
1 1 1

>>> B.x = 2

>>> print(A.x, B.x, C.x)
1 2 1

>>> A.x = 3

>>> print(A.x, B.x, C.x)
3 2 3
```



MRO: Method Resolution Order

- class B and C inherite from A
- all 3 classes have the same value of `x`
- `x` in B class, overrides the same property in A
- modifying `x` in A class, we have the same value for A and C classes

Remember about it ;)

Remember about it ;)
or compute:

```
L[A] = A O
L[B] = B A
L[C] = C A

L[C] = C + merge(A, O) = C A O
```

Remember about it ;)
or compute:

```
L[A] = A O
L[B] = B A
L[C] = C A

L[C] = C + merge(A, O) = C A O
```

or use `mro()` method:

```
A.mro()
[<class '__main__.A'>, <type 'object'>]
B.mro()
[<class '__main__.B'>, <class '__main__.A'>, <type
    ↪ 'object'>]
C.mro()
[<class '__main__.C'>, <class '__main__.A'>, <type
    ↪ 'object'>]
```

Exception handling

```
>>> try:
...     list_ = ['a', 'b']
...     int(list_[2])
... except ValueError, IndexError:
...     pass
```

Exception handling

```
>>> try:
...     list_ = ['a', 'b']
...     int(list_[2])
... except ValueError, IndexError:
...     pass

Traceback (most recent call last):
  File "<input>", line 3, in <module>
IndexError: list index out of range
```

Python 2

- old syntax is supported for backwards compatibility
- `except ValueError, e == except ValueError as e`
- `except ValueError, IndexError == except ValueError as IndexError`

Python 2

```
>>> try:
...     list_ = ['a', 'b']
...     int(list_[2])
```

Python 2

```
>>> try:
...     list_ = ['a', 'b']
...     int(list_[2])
...
... except (ValueError, IndexError) as e:
...     pass
```

Python 2

```
>>> try:
...     list_ = ['a', 'b']
...     int(list_[2])

... except (ValueError, IndexError) as e:
...     pass
```

Python 3

```
File "<stdin>", line 4
  except ValueError, IndexError:
```

Modifying list while iterating over it

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

Modifying list while iterating over it

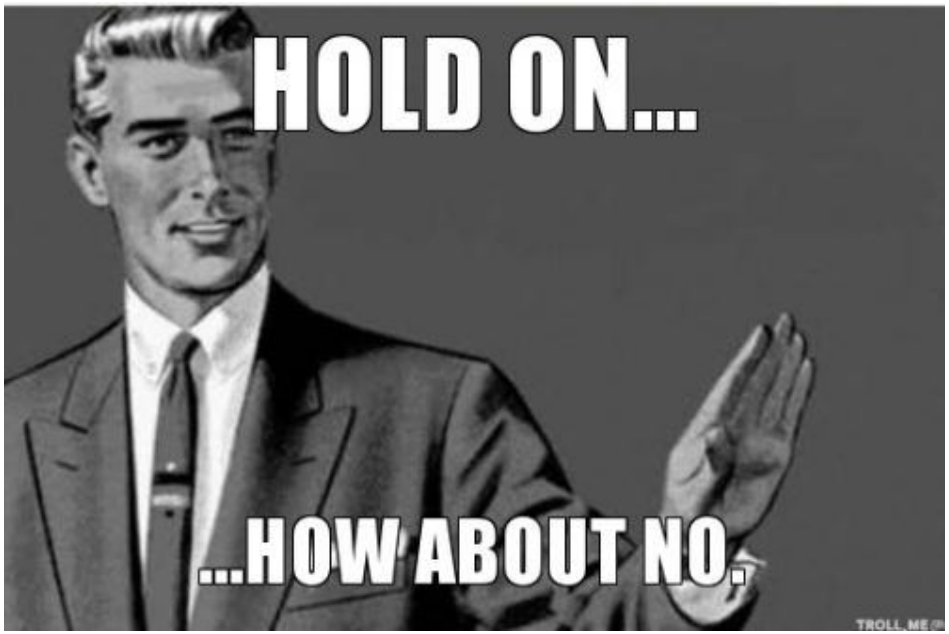
```
>>> odd = lambda x: bool(x % 2)
>>> numbers = list(range(10))
>>> for i in range(len(numbers)):
...     if odd(numbers[i]):
...         del numbers[i]
...

Traceback (most recent call last):
          File "<stdin>", line 2, in <module>
IndexError: list index out of range
```

Iterating

- iterate over the list
- remove odd values
- list is shrinking
- list is shorter than expected

```
>>> odd = lambda x: bool(x % 2)
>>> numbers = list(range(10))
>>> for n in numbers:
...     if odd(n):
...         numbers.remove(n)
...
>>> numbers
[0, 2, 4, 6, 8]
```

```
>>> odd = lambda x: bool(x % 2)
>>> numbers = list(range(10))
>>> for n in numbers:
...     if odd(n):
...         numbers.remove(n)
...
>>> numbers
[0, 2, 4, 6, 8]
```



```
>>> odd = lambda x: bool(x % 2)
>>> numbers
[0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8,
    ↪ 8, 9, 9]
>>> for n in numbers:
...     if odd(n):
...         numbers.remove(n)
...
>>> numbers
[0, 0, 1, 2, 2, 3, 4, 4, 5, 6, 6, 7, 8, 8, 9]
```

```
>>> odd = lambda x: bool(x % 2)
>>> numbers = list(range(10))
>>> numbers = [n for n in numbers if not odd(n)]
>>> numbers
[0, 2, 4, 6, 8]
```

Name clashing with Python Standard Library modules

```
app
|-sender.py
|-receiver.py
|-email.py
```

Name clashing with Python Standard Library modules

```
app
|-sender.py
|-receiver.py
|-email.py
```

sender.py

```
from email.mime.multipart import MIMEMultipart

msg = MIMEMultipart('alternative')
msg['Subject'] = "Link"
msg['From'] = 'from@email.com'
msg['To'] = 'to@email.com'
```

Name clashing with Python Standard Library modules

```
app
|-sender.py
|-receiver.py
|-email.py
```

sender.py

```
from email.mime.multipart import MIMEMultipart

msg = MIMEMultipart('alternative')
msg['Subject'] = "Link"
msg['From'] = 'from@email.com'
msg['To'] = 'to@email.com'
```

```
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ImportError: No module named mime.multipart
```

Standard Library

- import PSL `email`
- use functions from this module
- wonder, why is imported local, than expected one.

Python uses pre-defined order of importing modules. When we're trying to import `spam` it looks in:

1. built-in module (string, re, datetime, etc.)
2. searches for a file named `spam.py` in directories given by the variable `sys.path` in
   - the directory containing the input script (or the current directory).
   - PYTHONPATH (a list of directory names, with the same syntax as the shell variable PATH).
   - the installation-dependent default.

Bonus Mistake: Differences between Python 2 and Python 3

```
import sys

def bar(i):
    if i == 1:
        raise KeyError(1)
    if i == 2:
        raise ValueError(2)

def bad():
    e = None
    try:
        bar(int(sys.argv[1]))
    except KeyError as e:
        print('key error')
    except ValueError as e:
        print('value error')
    print(e)

bad()
```

Python 2

```
$ python foo.py 1
key error
1
$ python foo.py 2
value error
2
```

Python 2

```
$ python foo.py 1
key error
1
$ python foo.py 2
value error
2
```

Python 3

```
$ python3 foo.py 1
key error
Traceback (most recent call last):
  File "foo.py", line 19, in <module>
    bad()
  File "foo.py", line 17, in bad
    print(e)
UnboundLocalError: local variable 'e' referenced
    ↪ before assignment
```

When an exception has been assigned to a variable name using `as target`, it is cleared at the end of the except clause:

```
except E as N:
    try:
        foo
    finally:
        del N
```

```
import sys

def bar(i):
    if i == 1:
        raise KeyError(1)
    if i == 2:
        raise ValueError(2)
```

```
def good():
    exception = None
    try:
        bar(int(sys.argv[1]))
    except KeyError as e:
        exception = e
        print('key error')
    except ValueError as e:
        exception = e
        print('value error')
    print(exception)

good()
```

```
def good():
    exception = None
    try:
        bar(int(sys.argv[1]))
    except KeyError as e:
        exception = e
        print('key error')
    except ValueError as e:
        exception = e
        print('value error')
    print(exception)

good()
```

```
$ python3 foo.py 1
key error
1
$ python3 foo.py 2
value error
2
```

Thank you for your attention!