Robust Python Programs

EuroPython 2010

Stefan Schwarzer, SSchwarzer.com
info@sschwarzer.com

Birmingham, UK, 2010-07-20

Overview

- Introduction
- Indentation
- Objects and names
- Functions and methods
- Exceptions
- `exec` and `eval`
- `subprocess` module
- `for` loops
- Strings
- Optimization
- Tools for code analysis
- Summary

Introduction

- Python is a versatile language
- Concentration on the problem, not the language
- Compact solutions
- But: some mistakes occur frequently in Python programs
- Mainly by beginners and occasional programmers
- This talk (hopefully) describes the most important concepts, the most frequent errors and how to avoid them
- Talk discusses Python 2.x because it is commonly the default version on Posix systems

Introduction

Simplifications and Robustness

- Many points are, at first sight, more associated with “simplification” than with error prevention
- However, simplifications avoid more complicated code
- Code that is less complicated is easier to write and to read (important for subsequent changes)
- Simplifications may thus lead to more robust code
- But only if the code is easier to understand and not just shorter

Indentation

Basics

- Code blocks are denoted by the same indentation of the contained statements
- Indentation consists of “horizontal whitespace” (space and tab characters)
- Theoretically, both can be mixed—but should not
- If spaces and tabs are mixed, hard-to-spot program errors are possible
- But usually rather syntax errors because of inconsistent indentation
- For example, an `if` statement must be followed by indentation and an `except` clause must be preceded by “dedentation”

Indentation

Avoiding and Finding Problems

- Recommended: use exactly four spaces per indentation level
- See PEP 8, `http://www.python.org/dev/peps/pep-0008`
- Spaces often used automatically by editors if file ends with `.py`
- If not, configure the editor to insert four spaces if the tab key is pressed
- If you think you have indentation-related problems ...
- Make spaces and tabs visible in the editor, for example with `:set list` in Vim
- Use `find` and `grep`:

```
find .  -name "*.py" -exec grep -EnH "\t" {} \;
```

Identity Operator

- Checks if two objects are identical
- In other words, whether they are actually the same object
- In that case returns `True`, otherwise `False`
- The operator is the keyword `is`
- Identity is not the same as equality!

```
>>> 1 == 1.0
True
>>> 1 is 1.0
False
>>> [1] == [1]
True
>>> [1] is [1]
False
```

Names and Assignments
Basics

- Names (“variables”) do not contain objects in Python
- They refer (point) to objects
- `x = 1.0` binds the name `x` to the object `1.0`
- In an expression (for example on the right hand side of an assignment) a name stands for the object the name refers to

Names and Assignments

Immutable and Mutable Objects

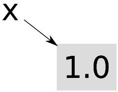
- Immutable objects usually have simple data types; examples are: `7.0`, `"abc"`, `True`
- Mutable objects are composite data, for example lists or dictionaries

```
>>> L = []
>>> L.append(2)
>>> L
[2]
>>> L[0] = 3
>>> L
[3]
```

Names and Assignments

Immutable Objects

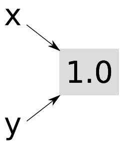
```
>>> x = 1.0
>>> y = x
>>> x is y
True
>>> y = 1.0
>>> x is y
False
```



Names and Assignments

Immutable Objects

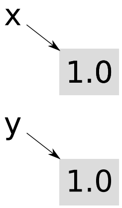
```
>>> x = 1.0
>>> y = x
>>> x is y
True
>>> y = 1.0
>>> x is y
False
```



Names and Assignments

Immutable Objects

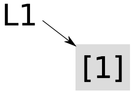
```
>>> x = 1.0
>>> y = x
>>> x is y
True
>>> y = 1.0
>>> x is y
False
```



Names and Assignments

Mutable Objects

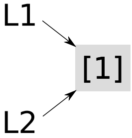
```
>>> L1 = [1]
>>> L2 = L1
>>> L1.append(2)
>>> L1
[1, 2]
>>> L2
[1, 2]
>>> L2 = [5, 6]
>>> L1.append(3)
>>> L1
[1, 2, 3]
>>> L2
[5, 6]
```



Names and Assignments

Mutable Objects

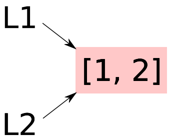
```
>>> L1 = [1]
>>> L2 = L1
>>> L1.append(2)
>>> L1
[1, 2]
>>> L2
[1, 2]
>>> L2 = [5, 6]
>>> L1.append(3)
>>> L1
[1, 2, 3]
>>> L2
[5, 6]
```



Names and Assignments

Mutable Objects

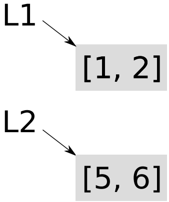
```
>>> L1 = [1]
>>> L2 = L1
>>> L1.append(2)
>>> L1
[1, 2]
>>> L2
[1, 2]
>>> L2 = [5, 6]
>>> L1.append(3)
>>> L1
[1, 2, 3]
>>> L2
[5, 6]
```



Names and Assignments

Mutable Objects

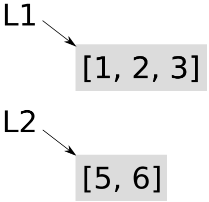
```
>>> L1 = [1]
>>> L2 = L1
>>> L1.append(2)
>>> L1
[1, 2]
>>> L2
[1, 2]
>>> L2 = [5, 6]
>>> L1.append(3)
>>> L1
[1, 2, 3]
>>> L2
[5, 6]
```



Names and Assignments

Mutable Objects

```
>>> L1 = [1]
>>> L2 = L1
>>> L1.append(2)
>>> L1
[1, 2]
>>> L2
[1, 2]
>>> L2 = [5, 6]
>>> L1.append(3)
>>> L1
[1, 2, 3]
>>> L2
[5, 6]
```



Names and Assignments

Combination of Immutable and Mutable Objects

```
>>> L = [1]
>>> t = (L,)
>>> t.append(2)
Traceback (most recent call last):
  File "<ipython console>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
>>> L.append(2)
>>> t
([1, 2],)
```

Names and Assignments

Combination of Immutable and Mutable Objects

```
>>> L = [1]
>>> t = (L,)
>>> t.append(2)
Traceback (most recent call last):
  File "<ipython console>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
>>> L.append(2)
>>> t
([1, 2],)
```

Names and Assignments

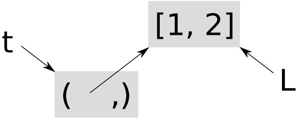
Combination of Immutable and Mutable Objects

```
>>> L = [1]
>>> t = (L,)
>>> t.append(2)
Traceback (most recent call last):
  File "<ipython console>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
>>> L.append(2)
>>> t
([1, 2],)
```

Names and Assignments

Combination of Immutable and Mutable Objects

```
>>> L = [1]
>>> t = (L,)
>>> t.append(2)
Traceback (most recent call last):
  File "<ipython console>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
>>> L.append(2)
>>> t
([1, 2],)
```



Comparisons

`is None` vs. `== None`

- `is` checks for identity, `==` for equality
- Recommended: `value is None`
- Reason: classes can modify the result of a comparison

```
>>> class AlwaysEqual(object):
...     def __eq__(self, operand2):
...         return True
>>> always_equal = AlwaysEqual()
>>> always_equal == None
True
>>> None == always_equal
True
>>> always_equal is None
False
```

Comparisons

"Trueness" and "Falseness"

- Of the built-in data types, numerical zero values (e. g. `0.0`), empty strings (`""`, `u""`), empty containers (`[]`, `()`, `{}`, `set()`, `frozenset()`), `None` and `False` are false.
  All other objects of built-in types are true.
- As a consequence, all these `if` conditions can be simplified:

```
if value == True        →   if value
if my_list != []        →   if my_list
if my_list == []        →   if not my_list
if len(my_list) == 0    →   if not my_list
if string == u""        →   if not string
```

  etc.

Comparisons

`if list` etc.

- What is so great about `if list` etc.? ;-)
- Shorter
- But more understandable (robust)?
- Yes—by rephrasing the condition
- Not "are values in this list?" but "are there any ...?"
- Example:

```
def show_names(names):
    if names:
        print "\n".join(names)
    else:
        print "no names"
```

Functions and Methods

Function Object Vs. Call

- Using a function (or method) without parentheses just gives us the function object
- ```
  fobj = open(filename, 'rb')
  # read first 100 bytes
  data = fobj.read(100)
  fobj.close
  ```
  ```

Functions and Methods

Function Object Vs. Call

- Using a function (or method) without parentheses just gives us the function object
- ```
  fobj = open(filename, 'rb')
  # read first 100 bytes
  data = fobj.read(100)
  fobj.close()  # call it!
  ```
  ```

Functions and Methods

Default Arguments

- Default arguments are only evaluated upon the definition, i. e. when the function or method is parsed and compiled
- Not upon each call
- ```
  >>> def append_to_list(obj, L=[]):
  ...     L.append(obj)
  ...     return L
  ...
  >>> append_to_list(2)
  [2]
  >>> append_to_list(5)
  [2, 5]
  ```
  ```

Functions and Methods

Names in a Call

- In a call of a function or method the argument names can be written explicitly
- Therefore the order of the arguments in a call can be different from their order in the definition
- The following calls are equivalent:

```
>>> def f(a, b, c):
...     return [a, b, c]
...
>>> f(1, 2, 3)
[1, 2, 3]
>>> f(a=1, b=2, c=3)
[1, 2, 3]
>>> f(b=2, c=3, a=1)
[1, 2, 3]
```

Functions and Methods

Arguments "Passed Through"

- Passing arguments "through" a function can be useful
- ```
  >>> def f(a, b, c):
  ...     print a, b, c
  ...
  >>> def g(*args, **kwargs):
  ...     print "Positional arguments:", args
  ...     print "Keyword arguments:", kwargs
  ...     f(*args, **kwargs)
  ...
  >>> g(1, c=3, b=2)
  Positional arguments: (1,)
  Keyword arguments: {'c': 3, 'b': 2}
  1 2 3
  ```
  ```

Functions and Methods

Passing Arguments by Name Binding

- Passing an argument works like an assignment
- Name is attached to an object
- ```
  >>> def delete_list(list_):
  ...     "Delete all elements from the list."
  ...     list_     = []  # new local name
  ...
  >>> a_list = [1, 2, 3]
  >>> delete_list(a_list)
  >>> a_list
  [1, 2, 3]  # no change!
  ```
  ```

Functions and Methods

Passing Arguments by Name Binding

- Passing an argument works like an assignment
- Name is attached to an object
- ```
  >>> def delete_list(list_):
  ...     "Delete all elements from the list."
  ...     list_[:] = []  # changed argument in-place
  ...
  >>> a_list = [1, 2, 3]
  >>> delete_list(a_list)
  >>> a_list
  []         # now changed
  ```
  ```

Exceptions

Why Exceptions?

Error handling in some languages (Shell, C, . . .) is done with error codes

Possible problems with error codes:

- Error handling makes return values and thus their handling more complex (e. g. using a tuple instead of a simple type)
- Error codes may have to be "passed down" a long call chain
- If a check for an error code is forgotten, undefined consequences occur, maybe to be noticed only much later

Exceptions

Missing or Too Generic Exception Class

- ```
  try:
      # do something ...
  except:
      # error handling
  ```
- Same issue with `except Exception:`
- Problem: some exceptions are caught unintentionally (`NameError`, `AttributeError`, `IndexError`, ...)
- This easily masks programming errors
  ```

Exceptions

Missing or Too Generic Exception Class

- ```
  try:
      # do something ...
  except:
      # error handling
  ```
- Same issue with `except Exception:`
- Problem: some exceptions are caught unintentionally (`NameError`, `AttributeError`, `IndexError`, ...)
- This easily masks programming errors
- ```
  try:
      fobj = opne("/etc/passwd")
      ...
  except:
      print "File not found!"
  ```

Exceptions

Missing or Too Generic Exception Class

- ```
  try:
      # do something ...
  except:
      # error handling
  ```
- Same issue with `except Exception:`
- Problem: some exceptions are caught unintentionally (`NameError`, `AttributeError`, `IndexError`, ...)
- This easily masks programming errors
- ```
  try:
      fobj = opne("/etc/passwd")
      ...
  except:
      print "File not found!"
  ```

Exceptions

Missing or Too Generic Exception Class

- ```
  try:
      # do something ...
  except:
      # error handling
  ```
- Same issue with `except Exception:`
- Problem: some exceptions are caught unintentionally (`NameError`, `AttributeError`, `IndexError`, ...)
- This easily masks programming errors
- ```
  try:
      fobj = opne("/etc/passwd")
      ...
  except:
      print "File not found!"
  ```
- List of exception classes at `http://docs.python.org/library/exceptions.html`

Exceptions

Too Much Code in the `try` Clause

```
def age_from_db(name):
    ...

try:
    person[name][age] = age_from_db(name)
except KeyError:
    print 'No record for person "%s"' % name
```

Exceptions

Too Much Code in the `try` Clause

```
def age_from_db(name):
    return cache[name]

try:
    person[name][age] = age_from_db(name)
except KeyError:
    print 'No record for person "%s"' % name
```

Exceptions

Too Much Code in the try Clause

```
def age_from_db(name):
    return cache[name]

# do not mask possible exception
db_age = age_from_db(name)
try:
    person[name][age] = db_age
except KeyError:
    print 'No record for person "%s"' % name
```

Exceptions

Freeing Resources

- Make sure there are no resource leaks:

```
db_conn = connect(database)
try:
    # database operations
    ...
finally:
    db_conn.rollback()
    db_conn.close()
```

- Since Python 2.5 the `with` statement can be used for files and sockets

```
from __future__ import with_statement # for Py 2.5
with open(filename) as fobj:
    data = fobj.read()
# file after ‘with‘ statement automatically closed
```

Exceptions

Multiple Exceptions in One `except` Clause

```
try:
    # can raise ValueError or IndexError
    ...
except ValueError, IndexError:
    # error handling for ValueError and IndexError
    ...
```

Exceptions

Multiple Exceptions in One `except` Clause

```
try:
    # can raise ValueError or IndexError
    ...
except ValueError, IndexError:
    # error handling for ValueError and IndexError
    ...
```

Problem: without parentheses, `IndexError` in the error case actually is a `ValueError` object

Exceptions

Multiple Exceptions in One `except` Clause

```
try:
    # can raise ValueError or IndexError
    ...
except (ValueError, IndexError):
    # error handling for ValueError and IndexError
    ...
```

Problem: without parentheses, `IndexError` in the error case actually is a `ValueError` object

`exec` and `eval`

Problems

`exec` and `eval` interpret a string as Python code and execute it

Problems:

- Code becomes more difficult to read
- Indentation errors are more likely
- Syntax check is delayed until `exec`/`eval` is hit
- Prone to security flaws
- Limited code analysis by tools

exec and eval

Complex Code

- ```
  def make_adder(offset):
      # ensure consistent identation
      code = """
  def adder(n):
      return n + %s
  """ % offset
      exec code
      return adder

  new_adder = make_adder(3)
  print new_adder(2)  # 3 + 2 = 5
  ```
- ```
  def value_n(obj, n):
      return eval("obj.value%d" % n)
  ```

exec and eval

Avoiding Complex Code

- Include functions, classes etc. in other functions or methods

```
def make_adder(offset):
    def adder(n):
        return n + offset
    return adder

new_adder = make_adder(3)
print new_adder(2)  # 3 + 2 = 5
```

- Use getattr, setattr and delattr

```
def value_n(obj, n):
    return getattr(obj, "value%d" % n)
```

exec and eval

Security Flaws

- Example: Function plotter on a website

  Function plotter

  f(x) = 2*x + 3 Show

- ```
  def plot_function(func):
      points = []
      for i in xrange(-100, 101):
          x = 0.1 * i
          y = eval(func)
          points.append((x, y))
      plot(points)
  ```
  ```

exec and eval

Security Flaws

- Example: Function plotter on a website

  Function plotter

  f(x) = 2*x + 3 Show

- ```
  def plot_function(func):
      points = []
      for i in xrange(-100, 101):
          x = 0.1 * i
          y = eval(func)
          points.append((x, y))
      plot(points)
  ```

- Not a nice function:

  f(x) = os.system("rm -rf *") Show
  ```

exec and eval

Avoiding Security Flaws

- Check against valid values

  ```
  if input_ in valid_values:
      # ok
  else:
      # error (reject or use default)
  ```

  where `valid_values` may be a list or a set
- Use a parser for expressions (see function plotter example)
- May be difficult to write
- Some ready-made parsers in the PyPI (Python Package Index) or the Python Recipes (ActiveState)
- There are libraries which help write parsers (pyparsing, SimpleParse, PLY etc.); see `http://nedbatchelder.com/text/python-parsers.html`

The `subprocess` Module

- The `subprocess` module replaces some commands of the `os` module with safe variants
- ```
  import os
  def show_directory(name):
      return os.system("ls -l %s" % name)
  ```
- Ok for `name == "/home/schwa"`
- Not ok for `name == "/home/schwa ; rm -rf *"`
- Sanitizing of such strings is difficult and error-prone
- Better:

  ```
  import subprocess
  def show_directory(name):
      return subprocess.call(["ls", "-l", name])
  ```
- Also replacements for `os.popen` etc.
  ```

Loops

`for` Loops

- If the sequence in the `for` loop is empty, the loop's body is not executed at all
- Iterate directly over sequences, no index is necessary

```
languages = (u"Python", u"Ruby", u"Perl")
for i in xrange(len(languages)):
    print language[i]
```

Loops

for Loops

- If the sequence in the `for` loop is empty, the loop's body is not executed at all
- Iterate directly over sequences, no index is necessary

```
languages = (u"Python", u"Ruby", u"Perl")
for language in languages:
    print language
```

Loops

for Loops

- If the sequence in the `for` loop is empty, the loop's body is not executed at all
- Iterate directly over sequences, no index is necessary

```
languages = (u"Python", u"Ruby", u"Perl")
for language in languages:
    print language
```

- If indices are needed, use `enumerate`

```
languages = (u"Python", u"Ruby", u"Perl")
for index, language in enumerate(languages):
    print u"%d: %s" % (index+1, language)
```

Strings

- Strings (both byte strings and unicode strings) are immutable
- `s.startswith(start)` checks if the string `s` starts with the string `start`; `endswith` checks at the end
- `substring in s` checks if `s` contains `substring`; `index` and especially `find` are unnecessary
- Negative indices count from the end of the string; Example: `u"Python talk"[-4:] == u"talk"`
- Here not discussed: byte strings vs. unicode strings, and encodings (important topics which are well worth a dedicated talk)
  `http://docs.python.org/howto/unicode.html`

Optimization

- Do not optimize while writing the code
- Generally does not lead to faster software
- Rather leads to code that is more difficult to maintain
- First develop clean code
- If it is too slow, use a profiler to find bottlenecks (`cProfile`/`profile` module)
- Limit optimization to the bottleneck you try to fix
- Revert “optimizations” which actually do not speed up the code
- More at `http://sschwarzer.com/download/optimization_europython2006.pdf`

Tools for Code Analysis

- They notice many of the discussed problems
- Not foolproof, but very helpful :-)
- PyLint
  `http://pypi.python.org/pypi/pylint`
  `http://www.logilab.org/project/pylint`
- PyChecker
  `http://pypi.python.org/pypi/PyChecker`
  `http://pychecker.sourceforge.net/`

Summary, Part 1/2

- Readability is more important than shortness
- Inconsistent indentation can be avoided easily
- Equality is not the same as identity
- There is no need to compare with empty lists, tuples etc. in conditional expressions
- Default arguments in functions are only evaluated once, during the function's definition
- In function calls, the order of named arguments is arbitrary
- Arguments can be "passed through" with `*args` and `**kwargs`
- To make changes to mutable objects visible outside a function, modify the argument itself, not just the name binding

Summary, Part 2/2

- Omit exception classes only in very special cases
- Limit the amount of code in a `try` clause
- Free resources with `try...finally` or `with`
- Put parentheses around multiple exception classes in `except` clauses
- `exec` and `eval` should be avoided if at all possible because they are prone to security flaws and other problems
- If calling out to a shell, do not use the `os` module but the `subprocess` module
- `for` loops rarely need an explicit sequence index
- Read how strings and encodings work
- Always use a profiler to optimize code—if you need to optimize at all. In any case, make the code work first.
- PyLint and PyChecker can help to write clean Python code

Thank You for Your Attention! :-)

Questions?

Remarks?

Discussion?