What's New in Python 2.2

LinuxWorld - New York City - January 2002

Guido van Rossum
Director of PythonLabs at Zope Corporation

guido@python.org
guido@zope.com
Overview

• Appetizers
  – Nested Scopes
  – Int/Long Unification
  – Integer Division

• First Course
  – Iterators
  – Generators

• Second Course
  – Type/Class Unification

• Dessert
  – Metaclas Programming
Nested Scopes

- **Motivation:**
  - functions inside functions need to access outer locals
- **Optional in 2.1**
  - from __future__ import nested_scopes
- **Standard in 2.2**
  - Can't turn it off (__future__ statement still allowed)
- **Only issue to watch out for:**
  - `def f(str):
      def g(x): print str(x)
      ...
  - In 2.1, `str` argument independent from `str()` function
  - In 2.2, `str` inside `g()` references outer `str`
Int/Long Unification

- $2^{1000}$ no longer raises OverflowError
  - but returns the appropriate long (no 'L' needed!)
- Ditto for:
  - $1000000 \times 1000000$
  - `sys.maxint + 1`
  - $10000000000000000000000000000000000$
- But:
  - $1 << 100$ is still zero
  - `repr(2**1000)` still ends in 'L'
  - `hex()`, `oct()`, `%x`, `%o`, `%u` differ for int vs. long
  - These will change in 2.3 or 3.0
Integer Division

• Motivation:
  – x/y computes different thing for ints than for floats
    • This is unique among operators, and confusing
  – def velocity(dist, time): return dist/time
    • Lack of type declarations makes this hard to debug

• Solution:
  – x//y new, "floor division": always truncates (to -Inf)
  – x/y unchanged, "classic division"
  – from __future__ import division
    • x/y returns "true division": always a float value

• In Python 3.0, x/y will be true division
  – Use Tools/scripts/fixdiv.py tool for conversion help
Iterators

• Motivation:
  – Generalized for loop innards, doing away with index

• Iterator abstracts **state** of for loop
  – Sequence may be a "real" sequence (e.g. a list)
  – Or a "virtual" sequence (e.g. the nodes of a tree)

• Most iterator calls are implied by for loops
  – For loop gets items one at a time from `next()`
  – Exception `StopIteration` signals end of sequence

• You *can* also call `next()` explicitly
Iterator API

- $t = \text{iter}(\text{obj})$ returns a (new) iterator for $\text{obj}$
  - Calls $t = \text{obj}.\text{__iter__}()$; $\text{obj}$ must provide $\text{__iter__}$
- $t.\text{next}()$ returns the next value
  - Raises $\text{StopIteration}$ when there is no next value
    - Alternatives rejected ($t.\text{end}()$, if $t$, $t.\text{next}() == \text{None}$)
- The iterator should be a separate object
  - Allow for multiple iterations over the same object
    - e.g. nested loops, multiple threads, nested calls
- But when $t$ is an iterator:
  - $\text{iter}(t)$ is $t$
  - This is handy with "for $x$ over $<\text{iterator}>$: ...")
The 'for' Loop

- for x in obj: f(x)
  is now translated (roughly) as follows:

- t = iter(obj)
  while 1:
    try:
      x = t.next()
    except StopIteration:
      break
  f(x)
Built-in Iterators

- for item in sequence: ... # nothing new :-)
- for line in file: ... # most efficient way!
- for key in dict: ... # must not modify dict!
  - for key, value in dict.iteritems(): ...
  - (unrelated but also new: if key in dict: ...)
- for val in iter(callable, endval): ...
  - while 1:
    val = callable()
    if val == endval: break
    ...
  - Example: iter(file.readline, "")
Generators

• Motivation:
  – Function to produce a sequence, one item at a time
    • State represented by local variables and/or PC
    • Using an object is overkill or inconvenient

• Example:
  – from __future__ import generators
  – def fibonacci(a=1, b=1):
    while 1:
      yield a
      a, b = b, a+b
  – t = fibonacci() # t is an iterator!
  – for i in range(10): print t.next()
How Does It Work?

- Stack frame is created in **suspended** state
  - Arguments in place, but no byte code executed yet
- t is a wrapper pointing to the suspended frame
  - t supports the iterator interface
- Calling t.next() **resumes** the stack frame
  - Execution continues where it left off previously
- A yield statement **suspends** the stack frame
  - Yield "argument" is the returned from t.next()
- An exception **terminates** the stack frame
  - Propagates out normally
  - Return (or falling through) raises StopIteration
Examples: "Iterator Algebra"

- def alternating(a):
  ta = iter(a)
  while 1:
    ta.next(); yield ta.next()

- def zip(a, b):
  ta = iter(a); tb = iter(b)
  while 1:
    yield (ta.next(), tb.next())

- for x, y in zip("ABC", "XYZ"):
  print x+y

- For a real-life example, see tokenize.py
Type/Class Unification

- Subclassing built-in types like dict or list
- "Cast" functions are now types, acting as factories
- Built-in objects have __class__, types have __dict__
- Overriding __getattr__() 
- Descriptors and the __get__() operation 
- property() 
- classmethod(), staticmethod() 
- super() and the new method resolution order (MRO) 
- Subclassing immutable types: __new__() 
- Performance hacks: __slots__ 
- Metaclasses
Subclassing Built-ins

- `class mydict(dict):
  def keys(self):
    K = dict.keys(self)
    K.sort()
    return K

- `class mylist(list):
  def __sub__(self, other):
    L = self[:]
    for x in other:
      if x in L: L.remove(x)
    return L

- These are "new-style" because of their base classes
- Note: `self` is an instance of the base class; the subclass is not a wrapper like UserDict or UserList
"Cast" Functions

- These built-ins are now types instead of factory functions (with the same signature):
  - `int`, `long`, `float`, `complex`
  - `str`, `unicode`
  - `tuple`, `list`
  - `open` (now an alias for `file`)
  - `type`

- These are new as built-in names:
  - `dict`, `file`
  - `object`: the universal base class (new-style)

- Useful new idiom: `if isinstance(x, file)` etc.
Unified Introspection

- obj.__class__ == type(obj)
  - Exception for unconverted 3rd party extension types
- Types have __dict__, __bases__
- All methods and operators shown in __dict__
  - E.g. list.__dict__ contains 'append', '__add__' etc.
  - list.append is an "unbound method":
    - list.append(a, 12) same as a.append(12)
    - list.append.__doc__ yields the doc string
- list.__bases__ == (object,)
- list.__class__ is type
Overriding `__getattribute__`

- class mylist(list):
  def __getattribute__(self, name):
    try:
      return list.__getattribute__(self, name)
    except AttributeError:
      return "Hello World"

- The `__getattribute__` method is always called
  - Not just when the attribute isn't found
  - Classic `__getattr__` also available on new-style classes

- Do not use `self.__dict__[name]`
  - Call the base class `__getattribute__` or `__setattr__"
Descriptors

- Generalization of unbound methods
  - Used by new-style object getattr, setattr
  - Also by classic instance getattr
- Descriptor protocol: `__get__()`, `__set__()`, `__delete()`
- `descr.__get__(object)` is **binding** operation
  - invoked by getattr when descriptor found in class
  - e.g. function or unbound method -> bound method
- `descr.__get__(None, class)`: *unbound* method
- `descr.__set__(object, value)`
  - invoked by setattr when descriptor found in class
- `__get__()` is also used by classic classes!
Properties

- class C(object):
  def get_x(self): return self.__x
  def set_x(self, value): self.__x = value
  x = property(get_x, set_x, doc="...")

- a = C()
- a.x # invokes C.get_x(a)
- a.x = 1 # invokes C.set_x(a, 1)
  - Descriptor overrides attribute assignment
- C.x.__doc__ == "...
- You can leave out set_x, or add del_x
class C:
    def spawn():
        return C()
    spawn = staticmethod(spawn)

c1 = C().spawn()
c2 = c1.spawn()

Use is just like in Java

Syntax is ugly, provisional
    - Python 2.3 may bring new syntax
Class Methods

- [Skip if running out of time]

- Similar to static methods, but get class arg:

  ```python
class C:
    def spawn(cls):
        return cls()

    spawn = classmethod(spawn)

  class D(C): pass
  c1 = C.spawn(); c2 = c1.spawn()
  d1 = D.spawn()
```
• class A(object):
  def save(self, f):
    "save state to file f"
    ...

• class B(A):
  def save(self, f):
    super(B, self).save(f)
    # instead of A.save(self, f)
    ...

• Motivation: see following slides
• Verbose syntax: may be fixed later
• Now it gets interesting:

• class C(A):
  def save(self, f):
    super(C, self).save(f)
    ...

• class D(B, C):
  def save(self, f):
    super(D, self).save(f)

• D().save(f)
  - D.save() -> B.save() -> C.save() -> A.save() !!!
  - This will be explained shortly :-)
• How does super(B, self).save(f) know to call C.save(f) when self is a D instance?!?!

• Answer: linearized MRO stored as D.__mro__
  – MRO = Method Resolution Order (see next slide)

• D.__mro__ == (D, B, C, A)

• super(B, self).save looks for B in self.__mro__ and looks for save in classes following it
  – It searches (C, A) for save
    • Specifically it looks for C.save and A.save in that order
Method Resolution Order

- Used for method lookup in new-style classes
- Compare to classic MRO:
  - classic MRO: left-to-right, depth-first (D, B, A, C, A)
  - new MRO removes duplicates from the left
- Motivation:
  - In diamond diagram, C.save should override A.save
    - If B and D don't define save, D.save should find C.save
  - This is more important because of 'object'
    - The universal base class for all new-style classes
Immutable Types

• Override `__new__` instead of `__init__`
• `__new__` is a static method with a class arg!
• `tuple(arg)` calls `tuple.__new__(tuple, arg)`
• `class mytuple(tuple):
  def __new__(cls, *args):
    return tuple.__new__(cls, args)`
• `t = mytuple(1, 2, 3)`
__slots__

• Allocates instance variables in the object structure instead of using a pointer to a __dict__; saves a lot of space

• class C(object):
  __slots__ = ['foo', 'bar']
  def __init__(self, x, y):
    self.foo = x; self.bar = y

• c1 = C(1, 2)
• c1.spam = 12 # error
Incompatibilities

• `dir()` behaves differently
  – shows instance variables and methods
  – shows methods from base classes as well
  – exceptions:
    • `dir(module)` returns only `__dict__` contents
    • `dir(class_or_type)` doesn't look in the metaclass

• `type('')).__name__ == "str" # was "string"
• `type(1L).__name__ == "long" # "long int"`
• class autosuper(type):
    def __init__(cls, name, bases, dict):
        attrname = '_%s__super' % name
        setattr(cls, attrname, super(cls))

• class myclass(object):
    __metaclass__ = autosuper
    def foo(self):
        self.__super.foo()
    ...etc...

• myclass.__class__ is autosuper
• Now or never :-)