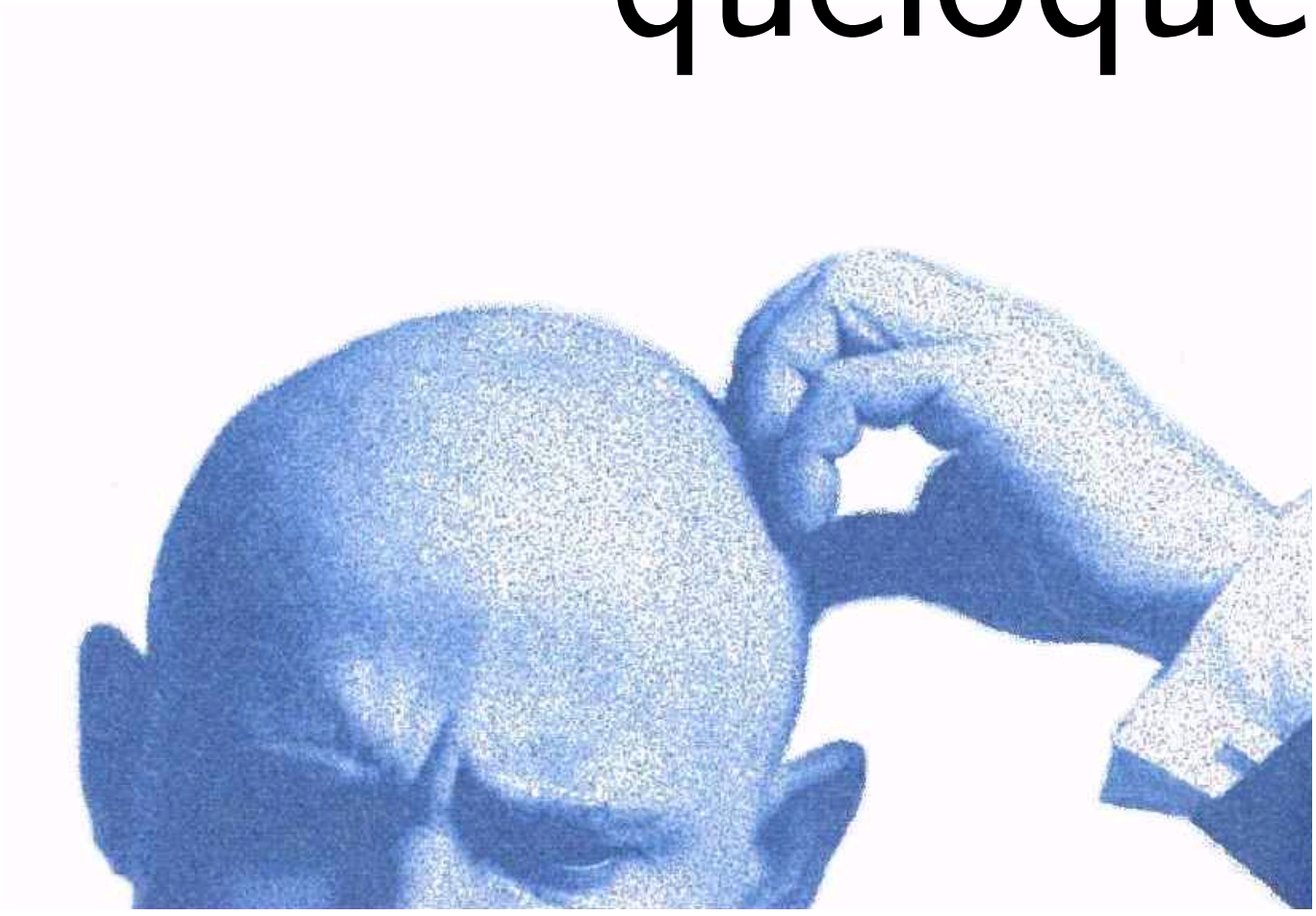
Python 3000

Facundo Batista

PyAr

Gracias especiales a **nessita** por su indispensable ayuda con LaTeXpara esta presentación

¿Python queloqué?



Evolución normal

- Versiones: Python x.y.z
  - x: mayor
  - y: menor
  - z: bugfix
- Normalmente no rompemos (casi) nada
  - warnings
  - from future
- Se barajaba una versión disruptiva desde hace años
- El futuro ya llegó

¿Por qué? ¿Para qué?

- Corregir errores de diseño, viejos principalmente
  - clases clásicas
  - división de enteros
  - print como declaración
- Los tiempos cambian: compromiso espacio/velocidad
  - str / unicode
  - int / long
- Nuevos paradigmas
  - views de los diccionarios
  - anotaciones en los argumentos

Nuevas funcionalidades y mejoras

Textos, textos, textos!

- En Py2: cadenas normales y unicode
  - no estaban bien separados los conceptos
  - confundía, sorprendía
- En Py3 tenemos bytes y texto
  - texto: nos abstraemos de la codificación
  - bytes: una secuencia de valores de 8 bits
  - son dos tipos de datos totalmente separados

Textos en Python 3

- Distinguimos ambos literales
  - b'...': bytes
  - '...': cadenas de caracteres (unicode)
  - u'...': también caracteres, por compatibilidad
- No hay conversiones implícitas
  - menos confusión, menos sorpresas
  - más fácil de seguir la Regla de Oro de Unicode
- Las fuentes tiene un default: UTF-8
  - menos confusión, menos sorpresas
  - no necesitamos más el `-*- coding:... -*-`
- `open()` tiene 'encoding' opcional!
  - como `codecs.open()`
- Unicode en los nombres
  - `>>> año = 3000`

Avistando diccionarios

- `dict.keys()`, `.values()`, `.items()`, ahora devuelven una *vista*
- Se actualizan cuando el diccionario cambia

```
>>> d = dict(a=3, b=4)
>>> v = d.values()
>>> v
dict_values([3, 4])
>>> list(v)
[3, 4]
>>> d["c"] = 8
>>> list(v)
[3, 8, 4]
```

- Son re-iterables, no se *gastan* como el viejo `.iteritem()`

```
>>> list(v)
[3, 8, 4]
>>> list(v)
[3, 8, 4]
```

Asterisquémonos

- Podemos tener argumentos sólo nombrados

```
>>> def foo(*args, bar=None):
...     print(args, bar)
>>> foo(1)
(1,) None
>>> foo(1, 2)
(1, 2) None
>>> foo(1, bar=7)
(1,) 7
```

- Asignación múltiple de largo variable

```
>>> a, b, *c = range(5)
>>> a, b, c
(0, 1, [2, 3, 4])
>>> a, *b, c = range(5)
>>> a, b, c
(0, [1, 2, 3], 4)
```

La función format()

- Integrada, formatea un sólo valor

```
>>> format(1.2321, "8.2f")
'    1.23'
```

- Cómo método de las cadenas

```
>>> "Son {0}: {val:.2f}".format("pesos", val=4)
'Son pesos: 4.00'
```

- Más poder, más flexibilidad
  - trabaja bien con los códigos de formateo conocidos
  - se puede definir el comportamiento del formato: `__format__`

```
>>> "{:,.2f}".format(Decimal(3123.5))
'3,123.50'
```

Abstract Base Classes (ABC)

- Nos permite definir clases abstractas con las cuales cumplir
- El módulo `collections` define estrictamente comportamientos

```
>>> dir(collections)
[..., 'Callable', 'Container', 'Hashable',
                      'Iterable', 'Sequence', ...]
```

- El chequeo rígido de interfaces es nuevo a Python
  - quizás dispare nuevas metodologías
  - con suerte no hará que aparezcan miles de controles `isinstance()`
  - el tiempo dirá si ayudan o lastiman al *duck typing*

Más y mejores números I

- Literales binarios y la función `bin()`

```
>>> 0b1001
9
>>> bin(34)
'0b100010'
```

- ¿Cuantos bits usa para un número?

```
>>> (27).bit_length()
5
```

- Fracciones!

```
>>> Fraction(1, 3)
Fraction(1, 3)
>>> print(Fraction(1, 3))
1/3
>>> print(Fraction(1, 3) * 3)
1
```

Más y mejores números II

- Nueva forma de buscar la representación más exacta de un `float`
  - `1.1` es `1.1000000000000008881784197001252323389053344726562 5`
  - antes se tomaban 17 decimales y ya: `'1.1000000000000001'`
  - ahora se busca el número más corto que siga siendo igual a lo almacenado: `'1.1'`
  - igual, lo subyacente no cambia

```
>>> 1.1 + 2.2 == 3.3
False
```

- Mejoras en Decimal
  - Conversión exacta desde `float`

```
>>> Decimal.from_float(1.1)
Decimal('1.100000000000000088878...0533447265625')
```

  - Está codeada (también) en C: de 12 a 120 veces más rápida

Interactuando con el sistema

- La E/S no depende más de `<stdio.h>` de C
  - independiente de la plataforma
  - unicode bien integrada en el stack subyacente
  - reescrito en C en 3.1, en 3.0 estaba en Python y era lento
  - nuevo flag 'x' para abrir un archivo: exclusivamente un archivo nuevo
- `os.sendfile()` provee 'zero-copy' entre archivos o sockets
- Todas las excepciones contra el sistema son `OSError`
  - Deprecamos `IOError`, `EnvironmentError`, `WindowsError`, `mmap.error`, etc...
  - No hace falta revisar el `errno` para saber qué pasó, hay excepciones puntuales como `FileNotFoundError`, `NotADirectoryError`, `PermissionError`, `TimeoutError`, etc...

Ejecuciones simultáneas

- Nuevo paquete `futures` para ejecutar código 'en paralelo'

```
PRIMES = [ lista de nros ]

def is_prime(n):
    (código para averiguar si n es primo)

with concurrent.futures.ProcessPoolExecutor() as executor:
    all_calls = executor.map(is_prime, PRIMES)
    for number, prime in zip(PRIMES, all_calls):
        print('%d is prime: %s' % (number, prime))
```

- Procesos (sensible cantidad default) o hilos

Mejoras en la sintaxis I

- Decoradores de clase
  - Fácil de aprender
  - Mucho más sencillo que metaclases

```
>>> def f(cls):
...     cls.log = lambda self, *a, **k: print(a, k)
...     return cls
...
>>> @f
... class C():
...     pass
...
>>> c = C()
>>> c.log(2)
(2,) {}
```

- Atrapando excepciones con menos riesgo
  - `except KeyError, err:` → `except KeyError as err:`
  - menos confuso: `except KeyError, AttributeError:`

Mejoras en la sintaxis II

- Declaración nonlocal

```
>>> b = 5
>>> def f():
...         b = 3
...         def g():
...             nonlocal b
...             b = 1
...         g()
...         print(b)
...
>>> f()
1
>>> print(b)
5
```

- Anotación de funciones
  - `def promedios(valores: list) -> float:`
  - se permite cualquier expresión
  - el intérprete no le da significado o propósito

Mejoras en la sintaxis III

- Literales para set, y *set comprehensions*

```
>>> {1, 2, 2, 3}
{1, 2, 3}
>>> {x**2 for x in (-1, 0, 1)}
{0, 1}
```

- También *dict comprehensions*!

```
>>> {y:x for x,y in d.items()}
{0: 'a', 1: 'b', 2: 'c'}
```

- Abriendo muchos archivos con `with`

```
>>> with open('mylog') as inp_f, open('out', 'w') as out_f:
...     for line in inp_f:
...         if '<critical>' in line:
...             out_f.write(line)
```

Más y mejores módulos

- Diccionarios ordenados
  - `collections.OrderedDict()`
  - Buenísimo para representar configuración
- Cache automático para funciones:
  - `@functools.lru_cache(maxsize=300)`
  - para pagar el costo una vez en funciones caras
- Acumulador

```
>>> list(itertools.accumulate([8, 2, 50]))
[8, 10, 60]
```

  - muy usado al integrar, o trabajando con probabilidady y estadística
- Y muchos más
  - `argparse`, para interpretar lineas de comandos, más completo y fácil
  - `faulthandler`, para crashes como un `Segmentation Fault`
  - `lzma`, soporte para `.xz` y `.lzma` en la biblioteca estándar!
  - Entornos virtuales, como `virtualenv`

Otras mejoras que no lastiman a nadie I

- Bytes mutables: `bytearray`

```
>>> bytearray(5)
bytearray(b'\x00\x00\x00\x00\x00')
>>> b = bytearray(range(5))
>>> b
bytearray(b'\x00\x01\x02\x03\x04')
>>> b[3] = 255
>>> b
bytearray(b'\x00\x01\x02\xff\x04')
```

- Entregando desde otro lado

```
>>> def g(x):
        for i in range(x):
            yield i
>>> def f(x):
        yield from range(x)
>>> list(g(5)), list(f(5))
[0, 1, 2, 3, 4], [0, 1, 2, 3, 4]
```

Otras mejoras que no lastiman a nadie II

- Nombres completos para clases y funciones

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.D.__name__
'D'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__name__
'meth'
>>> C.D.meth.__qualname__
'C.D.meth'
```

- No necesitamos argumentos para `super()`

Muchas mejoras no son compatibles

`print` es ahora una función

- El cambio es sencillo
  - `print 'pyar'` → `print('pyar')`
- Flexibilidad
  - fácil de reemplazar por tu propia función
  - más opciones, mejor legibilidad
  - `print(m, file=stderr)`
  - `print('\r' % avance, end='', flush=True)`

Iteradores/generadores FTW!

- La idea es tratar de usar siempre iteradores o generadores
- Es mucho más eficiente en velocidad y memoria
- Sólo es molesto en el intérprete interactivo
- Muchos cambios para ir hacia esto
  - `range()` ahora se comporta como `xrange()` (que no está más)
  - `map()` y `filter()`
  - `dict.items()`, `.keys()`, `.values()`
  - no están más `dict.iteritems()`, `dict.iterkeys()`, etc.
  - muchos son tipos de datos específicos, no realmente *iteradores*
- Tenemos un `next(it)` integrado
  - `.next()` → `.__next__()`

Ordenando ideas

- Chau al orden universal
  - no hay más comparaciones arbitrarias
  - `[2, 'w'].sort()` → `TypeError`
- `sorted()` y `list.sort()` no tienen más *cmp*
  - usar *key*
  - más limpio, más fácil
  - acordarse que el sort es estable!

```
>>> seq = [('b', 2), ('a', 2), ('b', 1), ('c', 1)]
>>> seq.sort(key=itemgetter(1))
>>> seq.sort(key=itemgetter(0))
>>> seq
[('a', 2), ('b', 1), ('b', 2), ('c', 1)]
```

- tampoco tenemos al *cmp* integrado

Pasamos la escoba

- Eliminamos `<>`
  - Usar `!=`
- Eliminamos `d.has_key(x)`
  - Usar `x in d`
- Eliminamos `reload()`
  - Usar `imp.reload()`
- Eliminamos `reduce()`
  - Usar `functools.reduce()`
- Eliminamos `apply()`
  - Usar `f(*args)`
- Eliminamos `basestring`
  - No tenemos más bytes y caracteres mezclados!
- Eliminamos los ``` ``
  - Usar `repr()`

Excepciones también como nuevas

- `raise 'foo' → TypeError`
- La sintaxis del `raise` es
  - `raise Exception(args)`
  - `raise Exception(args) from traceback`
- La sintaxis del `except` es
  - `except Exception:`
  - `except Exception1, Exception2:`
  - `except Exception as var:`

Seguimos cambiando I

- No más clases clásicas
  - Todas son *nuevas*
  - Cambió la sintáxis para indicar metaclases

```
>>> class C(metaclass=...):
```

- Enteros y no tanto
  - Los int y long ahora son el mismo objeto
  - División verdadera: `1/2` devuelve `0.5`
  - Nuevos literales para otras bases

```
>>> 0x101
257
>>> 0o101
65
>>> 0b101
5
```

Seguimos cambiando II

- Una biblioteca con menos polvo
  - Se eliminaron módulos viejos no mantenidos
  - Se armó un nivel más, es más intuitiva
- El import relativo ahora es explícito
  - `import foo` no es más dentro del paquete
  - `from .foo import bar`
- `None`, `True` y `False` ahora son palabras reservadas
  - `True = 0` → `SyntaxError`
- No tenemos más el desempaquetado de tuplas en los parámetros
  - `def f(a, (b, c)):` → `SyntaxError`
- `.__nonzero__()` → `.__bool__()`
- `raw_input()` → `input()`

Saltando a Python 3000

¿Ya ya ya ahorita?

- Python 2 está soportado, pero hay que ir pensando el cambio
  - 2.7 ya tiene un par de años, **y es el último**
  - algunos OSs ya no traen Python 2 por default
- Hay que migrar cuando ambas condiciones sean verdad:
  - que ya estés listo
  - que todas tus dependencias hayan sido portadas
- Aunque no migren ahora, vayan preparándose

La herramienta 2to3

- Es un traductor de código fuente, no mantiene contexto
- Maneja muy bien lo que es sintaxis
  - `print`; `<>`; `except E, v`:
- Maneja bastante bien a las funciones y los tipos integrados
  - `d.keys()`, `xrange()`, `apply()`
- No hace inferencia de tipos
- No sigue variables en el código

Preparandonos para Python 3000

- Migrar a Python 2.7, que es la versión de transición
  - muchos features de Py3 fueron incluídos aquí
  - tiene un flag `-3` que se queja de todo lo que se romperá luego
- Empezar a escribir código que sirva a futuro
  - no preocuparse por trivialidades que 2to3 soporta
  - como `callable()`, `<>`, etc.
- Focalizarse en lo que 2to3 no puede hacer
  - dejar de usar módulos obsoletos
  - comenzar a usar iteradores y generadores

Escribiendo código a futuro en 2.7

- Heredar las clases de `object()`
- Usar `dict.iterkeys()`, `.iteritems()`, etc.
- Usar `xrange()`, `sorted()` sin el *cmp*, `zip()`, etc
- Importar la división del futuro y usar `//` donde corresponda
- Heredar las excepciones de `BaseException`
- Usar `__lt__` y `__eq__` en lugar de `__cmp__`
  - si realmente necesitan la semántica de *cmp*: `(a > b) - (a < b)`
- Manejar texto de la mejor forma posible
  - aislar el uso de texto codificado
  - usar `b''` (en 2.7 sólo un alias) para todo lo explícitamente bytes
  - usar Unicode para todo lo que corresponda

Como recorrer el camino

- Plan de migración detallado
  - migrar a 2.7
  - comenzar a usar en 2.7 todo lo que se pueda de 3.3
  - ejecutar con `-3`, para descubrir qué podemos ir cambiando
  - convertir el código con 2-to-3
  - ejecutar todas las pruebas unitarias
- Cuando tengamos código Py3 válido
  - si se puede, pegar el salto!
  - si no: mantener dos versiones
  - o derivar código de Py3 desde Py2

Ejemplo de código de transición

- Teníamos en Py2 viejo:

```
print '2 ==', 11 / 5
```

- Vamos a Py2.7:

```
igualito
```

- Usamos el flag -3:

```
warning: floor division 11/5
```

- Corregimos 2.7:

```
print u'2 ==', 11 // 5
```

- 2to3 nos deja:

```
print('2 ==', 11 // 5)
```

Conclusión

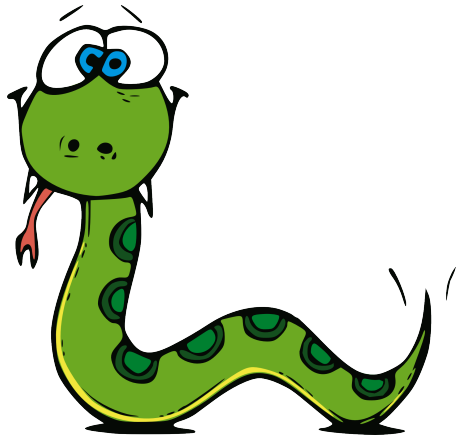
¿... entonces ...?

Conclusión

- Py3 es más limpio y liviano
- Hay menos trampas
  - Menos sorpresas
  - Menos casos particulares
  - Es más *cuadradito*
- No le tengas miedo a Python 3000
  - Divertite con las nuevas características
  - Pegar el salto no es trivial pero tampoco imposible

Sigue siendo

Python



(con todo lo que eso implica)

¡Muchas gracias!

¿Preguntas?

¿Sugerencias?

Facundo Batista

facundo@taniquetil.com.ar

http://www.taniquetil.com.ar

@facundobatista