Python 3000
il serpente cambia pelle?

di Giovanni Bajo
rasky@develer.com

develer

Cosa vedrete di bello (e brutto...)

- Perché Python 3000
  - c'era davvero bisogno?
- Anti-FUD
  - tutto quello che sai è falso!
- Principali novità
  - buy it now!
- Tecniche di migrazione
  - per non lasciarvi a piedi...

develer

Cosa vedrete di bello (e brutto...)

- Perché Python 3000
  - c'era davvero bisogno?
- Anti-FUD
  - tutto quello che sai è falso!
- Principali novità
  - buy it now!
- Tecniche di migrazione
  - per non lasciarvi a piedi...

develer

Python 2 e l'evoluzione

- Python è un linguaggio che tiene moltissimo alla retro-compatibilità.
  - Il *linguaggio* viene raramente modificato in modo da alterare programmi esistenti.
  - La *libreria* contiene moduli deprecati da anni.
  - La *API C* subisce semplici evoluzioni che tipicamente non richiedono troppo lavoro.

develer

Python 2: gli errori!

- Python non è perfetto
  - ... anche se manca poco :)
- Python viene utilizzato da 15 anni
- Sulla base dell'esperienza maturata sul campo, alcune scelte iniziali si sono rivelate sbagliate.
  - Ma non si possono correggere in Python 2!
- E quindi... Python 3000!

develer

Obiettivi di Python 3000

develer

Obiettivi di Python 3000

1) Correggere gli errori iniziali di progettazione

develer

Obiettivi di Python 3000

1) Correggere gli errori iniziali di progettazione
2) Correggere gli errori iniziali di progettazione

develer

Obiettivi di Python 3000

1) Correggere gli errori iniziali di progettazione
2) Correggere gli errori iniziali di progettazione
3) ... e anche correggere gli errori iniziali di progettazione.

develer

Obiettivi di Python 3000

1) Correggere gli errori iniziali di progettazione
2) Correggere gli errori iniziali di progettazione
3) ... e anche correggere gli errori iniziali di progettazione.

È davvero l'unico obiettivo!

develer

Ma serve davvero Python 3000?

- Python 3000 è l'occasione per rompere nettamente con il passato.
  - La prima volta in 16 anni di Python.
  - Chissà quando succederà di nuovo...
    - Guido dice: forse mai più
- A volte l'errore è stato già corretto aggiungendo la soluzione
  - Nel qual caso, via il vecchio!

develer

Non-obiettivi del Python 3000

- Riprogettare il linguaggio da zero.
  - Non solo ci vorrebbe troppo...
    - (devo proprio dirlo? Perl 6?)
  - ... ma non era neppure ***così sbagliato***!
- Implementare idee fantascientifiche senza alcuna esperienza pregressa sul campo.
  - Meta-programmazione, GC senza refcount, tipizzazione statica.

develer

Ma come? Io volevo le mega-feature!!!

- Le mega-feature sono quasi sempre ***aggiunte***.
  - E quindi possono attendere Python 3.1, 3.2, ...
  - Inoltre, molte feature possono essere prima testate sul campo tramite estensioni.
    - Es: decoratori.
- Solo cambiamenti di piccola e media entità.
  - Semplicemente un ***po' meglio***
  - ... e solo se vale la pena!
    - (secondo Guido)

develer

Quindi solo piccoli cambiamenti?

- Cambiamenti di piccola e media entità.
  - Niente di rivoluzionario
  - Semplicemente ***un po' meglio***.
- Solo cambiamenti che davvero ***valessero la pena***... secondo Guido.
  - Se non ne vale la pena, non c'è bisogno di cambiarlo per il gusto di cambiarlo.
    - C'è da migrare miliardi di linee di codice nel mondo!

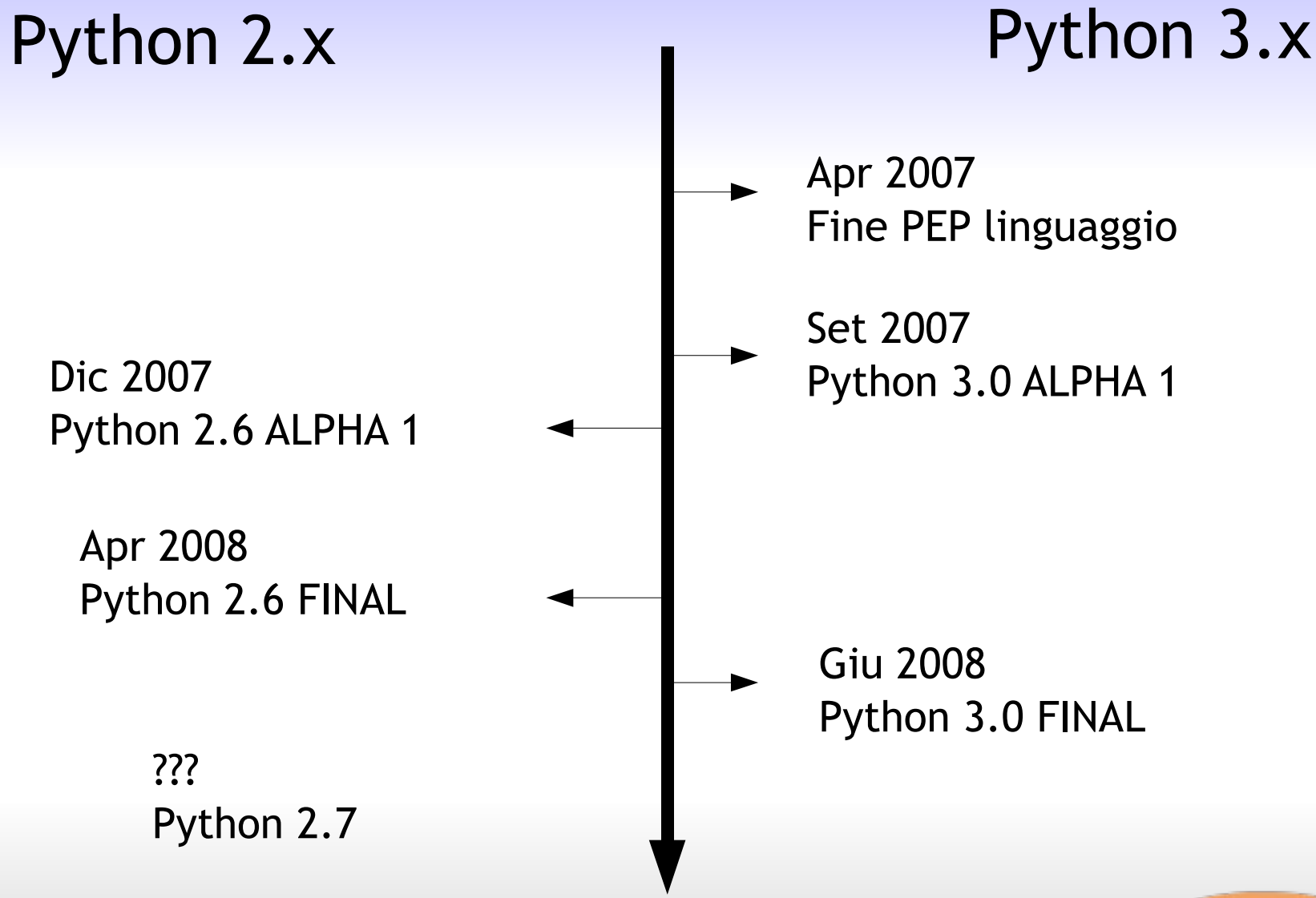
develer

Timeline

- Aprile 2007: Fine dei PEP linguaggio
  - Siamo in feature freeze!
- Settembre 2007: Python 3.0 ALPHA 1
- Dicembre 2007: Python 2.6 ALPHA 1
- Aprile 2008: Python 2.6 FINAL
- Giugno 2008: Python 3.0 FINAL
  - Python 2.7: ci sarà

develer

Timeline



develer

Cosa vedrete di bello (e brutto...)

- Perché Python 3000
  - c'era davvero bisogno?
- Anti-FUD
  - tutto quello che sai è falso!
- Principali novità
  - buy it now!
- Tecniche di migrazione
  - per non lasciarvi a piedi...

develer

FUD sul Python 3000

- Si è parlato tanto del Python 3000
  - Forse troppo...
- Prima di cominciare a parlare di quello ***che c'è***, parliamo di quello che ***non c'è***.
  - Giusto per sfatare un po' di miti!

develer

lambda

- Sono presenti in Python 3000. Sono ***identiche*** al Python 2.x.
  - Giuro, ***nessuna differenza***.
- Nessuna sintassi nuova per supportare statement multipli.
  - Tutte le proposte facevano schifo! Forse è davvero ***impossibile*** da integrare bene in Python.
- Nessun cambio di sintassi rispetto all'attuale.
  - Tutte le proposte facevano schifo quanto quella attuale, e a quel punto non vale la pena.

Tipizzazione statica

- Non ci sarà.
- C'è troppa poca esperienza sul campo per prendere la decisione giusta.
- Ma ci sarà una sintassi per decorare gli argomenti con dei tipi.
  - Nel caso qualcuno volesse ***sperimentare*** qualcosa nei prossimi anni.
  - `def f(x:int, y:bar(), z:[float]):`

develer

Garbage Collector

- Lo stesso di adesso.
- CPython è troppo legato all'implementazione corrente.
  - C'è stato un tentativo nel 2000 di sostituirlo, ma non è andato a buon fine (bassa performance).
- Inoltre, ci sono alternative: Jython, PyPy, IronPython...

Interfacce e funzioni generiche

- Non ci saranno.
- Troppa confusione, troppe idee
  - ABC: Abstract base classes
  - GF: Generic functions
  - ZI: Zope Interfaces
- Con piccolissime modifiche al linguaggio, si sperimenteranno come estensioni.
  - `__isinstance__`, `__issubclass__`

develer

Altre cose che NON cambieranno

- `self` rimane esplicito
- Niente meta-programmazione
- I literal dei container rimarranno
  - e anche `list()`, `dict()`, ecc.
- Il prompt interattivo rimarrà `>>>`
  - OK, chi se ne frega
    - (Io!)

develer

Cosa vedrete di bello (e brutto...)

- Perché Python 3000
  - c'era davvero bisogno?
- Anti-FUD
  - tutto quello che sai è falso!
- Principali novità
  - buy it now!
- Tecniche di migrazione
  - per non lasciarvi a piedi...

develer

Principali novità

- `print` diventa una funzione
- Viste sui dizionari
- Unificazione `int`/`long`
- Stringhe unicode e bytes
- Nuova libreria di I/O
- Identificatori unicode
- Nuova sintassi di formattazione stringhe
- ... e una marea di piccoli ritocchi.

develer

Print (PEP 3105)

- `print` aveva dei problemi:
  - Semantica strana in alcuni casi (soft-space)
    - `print "x\n", "y"`
  - Sintassi orrenda per ridirezione su file
  - Difficoltà di personalizzazione
    - Es: uso di tabulazioni per separare gli argomenti
  - Difficoltà di estensione
    - Per trasformare una `print` in un log, devo cambiare tutte le `print` in chiamate a funzione, una per una!

develer

Print (PEP 3105)

Python 2.x

- `print a, b`
- `print a, b,`
- `sys.stdout.write(str(a))`
- `print >>f, a, b`
- `print "\t".join(map(str,L))`

Python 3.x

- `print(a, b)`
- `print(a, b, end=" ")`
- `print(a, end="")`
- `print(a, b, file=f)`
- `print(*L, sep="\t")`

develer

Print (PEP 3105)

- Il vantaggio "percepito" di `print` è che tutti gli argomenti vengono implicitamente convertiti in stringa.
  - E questo rimane così com'è!
- Ma con Python 3.x possiamo:
  - Definire `def print()` in cima allo script
  - Aggiungere elegantemente nuove feature in futuro
    - Altri keyword argument

Viste sui dizionari (PEP 3106)

- I dizionari avevano una doppia sintassi:
  - `iterkeys()` / `keys()`, `iteritems()` / `items()`, `itervalues()` / `values()`
- Le versioni `iter*()` però richiedono spesso di fare copie:
  - Per iterare due volte o più
  - `set(a.keys()) == set(b.keys())`
  - `x in a.keys()`

develer

Viste sui dizionari (PEP 3106)

- Soluzione: le viste
  - Oggetti leggeri (reference al `dict`)
  - Sola lettura (immutabili dall'utente)
    - Ma rimangono validi e aggiornati se il dict muta
  - In pratica, "adattano" il dizionario ai vari usi.
- `.keys()` torna una vista simile ad un `set()`
  - `if a.keys() == b.keys()`
  - `for x in a.keys() & b.keys():`

develer

Viste sui dizionari (PEP 3106)

- `.items()` torna una vista simile ad un `set()` di tuple `(key, value)`
  - Solo `key` deve essere hashabile
  - `if (k,v) in d.items():`
- `.values()` torna una vista simile ad una collezione senza ordine e senza unicità
  - `x in d.values()` è O(n)
  - `a.values() == b.values()` è O(n^2)

develer

Unificazione int/long (PEP 237)

- Già iniziata dal Python 2.2
  - `10**10 == 10000000000L`
- Ora, esiste un solo tipo di intero: int
- Implementato internamente come long
  - Illimitato
- Un po' di trucchi per ottimizzare
  - ... ma non bastano: c'è ancora un 10% di performance da recuperare!

develer

Stringhe unicode

- Python 2.x: due tipi di "stringa"
  - `unicode`: utilizzata per testo Unicode
  - `str`: utilizzata per dati binari, e testo ASCII o codificato (latin-1, UTF-8, ...)
- Problemi:
  - `str` è ambigua
  - `str` per i dati binari è inefficiente e scomoda (immutabile)
  - Troppe funzioni per ragioni storiche tornano `str` invece che `unicode`

develer

Stringhe unicode

Python 2.x

- `unicode`
- `str` per testo
- `str` per binari
- `u"abc"`
- `"abc"`

Python 3.x

- `str`
- `str`
- `bytes`
- `"abc"`
- `b"abc"` o
  `bytes([97,98,99])`

develer

Stringhe unicode

- In Python 3.x, si usa `str` per tutto il testo
  - Es: i metodi per accedere ai file di testo tornano sempre str
  - Anche gli attributi nei `__dict__` delle istanze!
- `bytes` è un nuovo tipo di dato, mutabile
  - Ottimo per memorizzare dati binari
  - Ottimo per i protocolli di rete
  - Ha un subset di metodi da "stringa"
    - `find`, `index`, `startswith`, `replace`, ...

develer

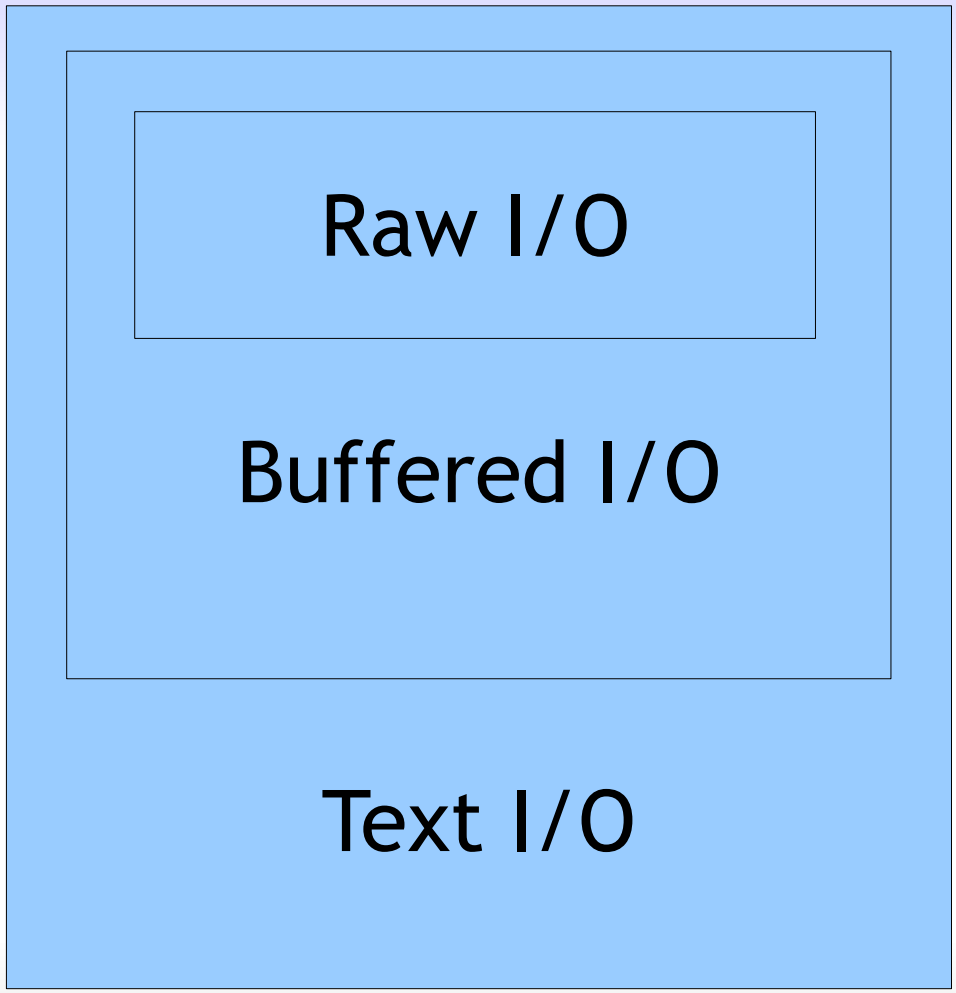
Nuovo I/O (PEP 3116)

- In Python 2.x, non esiste un vero e proprio modello di I/O
  - File-like: istanze con `read()` o `write()`
  - Funziona per usi semplici
  - Se mi arriva un file-like generico:
    - Come aggiungo una `readline()`?
    - Come decomprimo in streaming gzip/bzip2?

develer

Nuovo I/O (PEP 3116)

- Raw I/O
  - `read/write/seek/close`, check delle capabilities
  - `read` legge quanto può
  - `write` scrive quanto può
  - `readinto(bytes)`
- Buffered I/O
  - Come raw, ma gestisce buffering interno
- Text I/O
  - `readline`, encodings



develer

Nuovo I/O (PEP 3116)

- Struttura multi-livello, componibile
  - Per ora 3 livelli definiti.
- Ogni livello ***usa/adatta*** un'istanza del livello sottostante.
- ABC d'interfaccia già forniti
  - `io.RawIOBase`, `io.BufferedIOBase`, `io.TextIOBase`
- Qualche adaptor generico già pronto:
  - `BufferedReader`, `BufferedWriter`

develer

Nuovo I/O (PEP 3116)

- No panic!
  - Funziona comunque in duck typing
    - O anche no!
  - C'è comunque la funzione open()!
    - E gestisce anche gli encoding!
- Supporto primitivo per I/O non bloccante
  - Andrà migliorato...

develer

Identificatori Unicode (PEP 3131)

- Vediamo subito un esempio...

develer

Identificatori Unicode (PEP 3131)

```
def normpathϰ(pathϞ)Ϟ:
    """Normalize path, eliminating double slashes, etc."""
    if pathϞ == '':
        return '.'
    initial_slashesϠ = pathϞ.startswithϰ('/')Ϡ
    # POSIX allows one or two initial slashes, but treats three or more
    # as single slash.
    if (initial_slashesϠ and
        pathϞ.startswithϰ('//')Ϡ and not pathϞ.startswithϰ('///')Ϡ)Ϡ:
        initial_slashesℕ = 2
    compsϢ = pathϞ.splitϰ('/')Ϣ
    new_compsϢ = []Ϣ
    for compϞ in compsϢ:
        if compϞ in ('', '.')ⓣ:
            continue
        if (compϞ != '..' or (not initial_slashesℕ and not new_compsϢ)Ϡ or
             (new_compsϢ and new_compsϢ[-1]Ϟ == '..')Ϡ)Ϡ:
            new_compsϢ.appendϰ(compϞ)
        elif new_compsϢ:
            new_compsϢ.popϰ()Ϟ
    compsϢ = new_compsϢ
    pathϞ = '/'.join(compsϢ)Ϟ
    if initial_slashesℕ:
        pathϞ = '/'*initial_slashesℕ + pathϞ
    return pathϞ or '.'
```


develer

Identificatori Unicode (PEP 3131)

- OK, era uno scherzo! :)
- Questo però no:

```
import random
満は = range(100)
random.shuffle(満は)
末 = 満は.pop(7)
if len(末) > 58:
    print "ラーになる!!!"
```


develer

Identificatori Unicode (PEP 3131)

- Vantaggi (pochi):
  - Insegnamento nelle scuole elementari e medie
  - Possibilità di esprimere correttamente concetti nella lingua nativa
- ***No-panic*** mode:
  - La stdlib rimane solo ASCII e solo inglese (PEP 8)
    - La comunità Python idem (CheeseShop, etc.)
  - In Java e C# ha funzionato

develer

Formattazione di stringhe (PEP 3101)

- In Python 2.x, ci sono 2 diversi metodi:
  - operatore `%`
  - `string.Template` (poco diffuso)
- La sintassi di `%` è troppo rigida:
  - O posizionali, o keyword
  - Incoerenza singolo argomento
  - Eredità del C (il tipo non servirebbe)
    - Ma `%x` o `%r` sì!

develer

Formattazione di stringhe (PEP 3101)

- Nuova sintassi:

```
"Mi chiamo {0} e siamo al {conf}".format("Gio", conf="Pycon")
```

- Vantaggi:
  - Supporto per argomenti posizionali e per keyword
  - Posizionali espliciti (i18n-friendly)
  - Nuove potenzialità

develer

Formattazione di stringhe (PEP 3101)

- Compound fields
  - `"{0.name}","{0[key]}", "{0.{1}}"`
- Conversion specifiers
  - Fill, align, width, precision, type
  - `"{0:-.4g}","{0:x}","{0:#^20}"`
- Custom conversion specifiers
  - `"{0:H:M:S}".format(datetime.now())`

develer

Piccoli ritocchi #1

- True division di default
  - `3 / 2 = 1.5`
- Addio classic classes
- Import assoluti di default
  - Import relativi espliciti.
- `xrange()` diventa `range()`
- Unpacking esteso di iterabili:
  - `a,*b = (1,2,3,4)  # b = (2,3,4)`
  - `a,*b,c = (1,2,3,4)  # b = (2,3)`

develer

Piccoli ritocchi #2

- *`except E1, E2 as e`*
  - Invece che `except (E1, E2), e`
- Sintassi unica per eccezioni
  - `raise "addio!"`
  - `raise Exception, "addio"`
- Rimossa sintassi alternativa per `repr(x)`
  - `` `x` ``
    - E l'uso del backtick è stato bandito per sempre!

develer

Piccoli ritocchi #3

- Impossibile ordinare variabili di tipi diversi
  - `[1, "abc", None].sort()`
  - `<`, `<=`, `>`, `>=` sollevano `TypeError`
  - `==` e `!=` invece funzionano sempre, ovviamente!
- Argomenti solo per keyword:
  - `def foo(a, b, c, *args, x, y)`
  - `def foo(a, b, c, *, x, y)`
- Addio parametri-tuple nelle funzioni:
  - `def foo(a, (b, c), d)`

develer

Piccoli ritocchi #4

- Sintassi per set letterali
  - `{0, 1, 2} == set([0, 1, 2])`
  - Ma: `{} == dict() != set()`
- Supporto per name binding esterno:

```
x = 0
def foo():
    nonlocal x
    x = 1
foo()
assert x == 1
```


develer

Cosa vedrete di bello (e brutto...)

- Perché Python 3000
  - c'era davvero bisogno?
- Anti-FUD
  - tutto quello che sai è falso!
- Principali novità
  - buy it now!
- Tecniche di migrazione
  - per non lasciarvi a piedi...

develer

OK, lo compro! E ora?

- Ci sarà un piano di migrazione preciso
  - Le discussioni sono ancora in corso
  - C'è volontà di aiutare la comunità.
- Python 3000 rompe col passato. Quindi:
  - Vietato conservare compatibilità 2.x
    - Ma non si cambiano le cose per il gusto di farlo
  - Ma anche: vietato ingolfare 2.x di roba 3.x
    - Si rovina la base di codice

develer

1° regola: niente intersezioni!

- Sbagliato cercare l'intersezione magica
  - Cioè codice valido 2.x ***e*** 3.x
- Anche se esiste, è troppo limitante:
  - Sarebbe codice brutto sia per 2.x, sia per 3.x
  - Si perde il meglio di entrambe le versioni
  - Si ricorrerebbe agli if per salvarsi

develer

Il trio dei "traghettatori"

1) Tool di refactoring: **2to3**
   - Framework pronto, supporta già molti pattern.
   - Focus su correttezza: l'output non è perfetto, ma funziona.
2) **Backport** di ciò che non sporca troppo 2.x:
   - `from __future__ import \`
     `fantastic_py3k_feature`
3) Opzione magica **`python2.x -3`**:
   - Avverte sui costrutti dove 2to3 non funzionerà!

develer

Cosa sa fare 2to3 (esempi):

Python 2.x	Python 3000
`apply(fun, args, kwds)`	`fun(*args, **kwrds)`
`d.iterkeys()`	`d.keys()`
`print >>sys.stderr, x,`	`print(x, end=" ", file=sys.stderr)`
`d.has_key(x)`	`d in x`
`exec a in b,c`	`exec(a, b, c)`

develer

Checklist per migrazione

1) Aggiungere ed usare tutti i future
   - `from __future__ import unicode_literals`
2) Lanciare la propria testsuite con `python -3`
3) Convertire il codice con `2to3`
4) Ripulire e migliorare il codice auto-convertito
5) Enjoy Python 3000!

develer

Checklist per doppia versione

1) Aggiungere ed usare tutti i future
   - `from __future__ import unicode_literals`
2) Lanciare la propria testsuite con `python -3`
3) Convertire il codice con `2to3`
4) Ripulire e migliorare il codice auto-convertito
5) Enjoy Python 3000!

develer

Conclusioni

develer

Conclusioni

- Python 3000: corregge i peccati originali

develer

Conclusioni

- Python 3000: corregge i peccati originali
- Il "cambio di pelle" è solo un bel lavaggio con smacchiante e candeggina
  - Una fregatura, insomma!

develer

Conclusioni

- Python 3000: corregge i peccati originali
- Il "cambio di pelle" è solo un bel lavaggio con smacchiante e candeggina
  - Una fregatura, insomma!
- E' sempre lo stesso serpente, solo tirato più a lucido e ben pettinato
  - E a noi, del resto, piace così!

develer

Domande?

(facili, per favore...)

develer