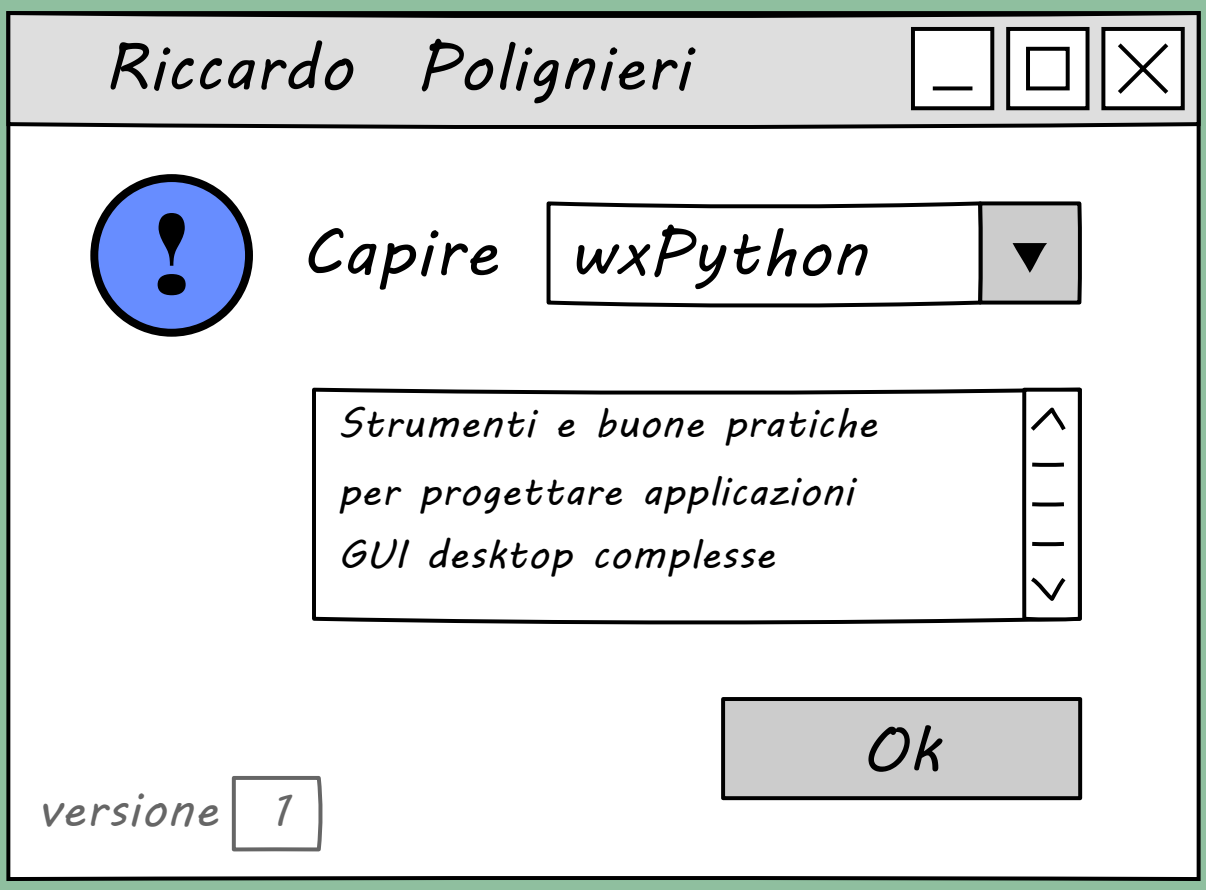
Riccardo Polignieri
Capire
wxPython
Strumenti e buone pratiche
per progettare applicazioni
GUI desktop complesse
Ok
versione
1

Riccardo Polignieri

Capire wxPython

Strumenti e buone pratiche per progettare applicazioni GUI desktop complesse

versione 1 – maggio 2019

Questo libro è in vendita su Leanpub: https://leanpub.com/capirewxpython

CAMPIONE DEL LIBRO PER VALUTAZIONE - https://leanpub.com/capirewxpython

Indice

Parte I

Introduzione.

1 Introduzione.

Benvenuti! Questo libro è l'evoluzione di una raccolta di *Appunti wxPython* che ho iniziato nel 2012. Dopo qualche anno di interruzione, quando nel frattempo anche lo sviluppo di wxPython era stagnante e il supporto a Python 3 tardava ad arrivare, mi sono finalmente deciso a riprendere in mano il lavoro adesso che la nuova versione di wxPython *Phoenix* è finalmente uscita. Tornare a scrivere gli *Appunti* si scontrava però con un problema di fondo: ormai quella documentazione era divenuta obsoleta, piena di codice Python 2 non più aggiornato. Per questo ho rivisto a fondo tutto quello che avevo scritto in passato, cambiando radicalmente molte parti. Ho poi aggiunto diversi capitoli nuovi, ampliato gli indici, rifatto e testato il codice, limato i dettagli… fino a raggiungere uno stato che mi sembrava «abbastanza pronto» per essere presentato!

Il bello di pubblicare in proprio e «virtualmente», senza i limiti imposti dalla casa editrice tradizionale e dal formato cartaceo, è che nessuno ti costringe a tagliare nulla. In questo libro mi prendo la libertà di approfondire, spiegare, esemplificare *a fondo* anche le parti più esotiche che un libro di carta sarebbe costretto a lasciar cadere per comprensibili vincoli economici. Riconosco che in alcuni punti potrei aver esagerato (ehm, i capitoli sugli eventi…): il fatto è che non volevo scrivere l'ennesimo libro che si limita a «mettere in bella copia» le parti facili che potete comunque trovare su Internet, e poi vi lascia a piedi quando le cose cominciano a farsi difficili.

Questo libro non si rivolge a principianti assoluti di Python e della programmazione: ho fatto del mio meglio per spiegare in modo chiaro e piano, ma dovete lo stesso sapere almeno vagamente che cosa è una classe e come si lavora con gli oggetti in Python. Del resto wxPython è un framework complesso, che richiede già qualche esperienza per essere maneggiato. In ogni caso, rispetto agli *Appunti* originali, che erano più sintetici, ho diluito la spiegazione di certi passaggi più tecnici e soprattutto ho aggiunto un *tutorial* introduttivo che dovrebbe semplificare le cose anche per il principiante.

Un altro grande vantaggio della pubblicazione indipendente è che posso pubblicare il libro man mano che viene scritto: grazie al sistema di Leanpub, chi compra il libro oggi avrà la possibilità di scaricare gratuitamente le versioni aggiornate con i nuovi capitoli, man mano che arriveranno… e ne arriveranno eccome! Considerate che in questo momento il libro è già lungo più di 300 pagine ma non contiene che un terzo delle cose che ho in mente di scrivere.

I miei *Appunti* costituivano già all'epoca il manuale più completo disponibile in Italiano su wxPython. Questo libro fin dalla sua prima versione amplia gli *Appunti* di un buon venti per cento, e continuerà a crescere. Ora, dato lo stato non proprio brillantissimo dell'editoria informatica nostrana, credo che si possa dire senza timore di smentita: non uscirà *mai* una guida più completa di questa in Italia. Se considerate poi che wxPython è un framework maturo e molto stabile, questo libro non è destinato a invecchiare tanto presto. Se siete interessati alla programmazione di interfacce grafiche con Python, questo dovrebbe essere il libro che fa per voi.

Importante: ...e a proposito: per favore *non piratate questo libro*. Questo libro non è scritto da un miliardario e pubblicato da una multinazionale. Non è un best-seller planetario a base di draghi e di troni. Questo libro è curato in modo artigianale e si rivolge a un pubblico di nicchia, molto di nicchia; costa incredibilmente poco rispetto al suo volume e (se posso permettermi) al suo contenuto. Piratare questo libro vuol dire distruggere irreparabilmente il lavoro che ci sta dietro e annullare la possibilità di estenderlo in futuro. Per favore, compratelo e basta. Grazie di cuore.

1.1 Convenzioni usate nel libro.

Questa è una sezione «obbligatoria» in tutti i libri di informatica, ma davvero non c'è molto da dire qui. In primo luogo, il libro è in Italiano: il codice degli esempi (nomi delle variabili, etc.) è però in Inglese. Non è mai una buona idea scrivere il codice in qualsiasi altra lingua che non sia l'Inglese e questo libro non vuole dare il cattivo esempio al lettore. L'unica licenza che mi sono concesso è di mantenere in Italiano i commenti nel codice.

Il codice di questo libro *non* segue il *code style* tipico di Python (la PEP 8, per intenderci), per il semplice motivo che neanche wxPython lo segue: in wxPython i nomi delle funzioni sono «CamelCase» perché derivano dalla consuetudine C++ di wxWidgets.

I nomi delle funzioni sono sempre riportati per intero nel testo: per esempio, sempre `wx.Classe.Funzione` e mai solo `Funzione`, anche quando sarebbe ovvio dal contesto. Questo è più verboso ma anche più chiaro e aiuta la ricerca negli indici. Solo i capitoli del tutorial riportano i nomi in forma abbreviata, per agevolare la lettura. Nel capitolo sulla *gerarchia delle classi* (pagina 54) questo criterio è descritto in modo più approfondito.

In genere gli esempi di codice sono eseguibili: per brevità omettono solo la dichiarazione `import wx` iniziale e la clausola abituale

```
if __name__ == '__main__':
    app = wx.App(False)
    MainFrame(None).Show()
    app.MainLoop()
```

che funziona da *entry-point*: potete aggiungere queste righe al codice degli esempi per renderli funzionanti. Una spiegazione più dettagliata del significato di questa clausola *verrà data* (pagina 58) nel capitolo dedicato alla `wx.App`.

wxClassic: Questo libro è stato scritto per la nuova versione del framework (wxPython *Phoenix*) e Python 3. Per chi avesse già esperienza con le vecchie versioni che supportavano Python 2 (dette oggi «wxPython Classic»), o per chi magari è impegnato a migrare una vecchia applicazione, le differenze più importanti sono riportate in riquadri come questo.

2 Piano dell'opera e aggiornamenti.

Questo libro è un *work in progress*: periodicamente saranno pubblicate nuove versioni con aggiunte e variazioni.

versione 1, maggio 2019

- **Introduzione**, Piano dell'opera, Installazione e documentazione
- **Tutorial** in 10 parti
- **Fondazione di un programma wxPython**: La gerarchia delle classi di wxPython; `wx.App`, il motore di un programma wxPython; La catena dei «parent»; Gli Id in wxPython; Bitmask e flag di stile; Finestre, dialoghi, panel: contenitori wxPython; I colori in wxPython; I font in wxPython; Interfacce comuni dei widget
- **Elementi di uso comune**: Selettori, offrire una scelta tra più opzioni; I sizer - prima parte; I sizer - seconda parte; Le dimensioni in wxPython; Questioni varie di stile
- **Il sistema degli eventi**: Gli eventi, le basi da sapere; La propagazione degli eventi; Collegare gli eventi; Eventi personalizzati; Tecniche per gli eventi; Handler personalizzati; Pattern Publish/Subscribe; Eventi di chiusura; I timer; GUI non-bloccanti e thread; Altre tecniche per le GUI non-bloccanti
- **Altri strumenti di uso comune**: I menu - prima parte; I menu - seconda parte; La barra degli strumenti e di stato
- **Widget di uso non comune**: Layout con i contraints
- **Strumenti di programmazione avanzata**: I validatori - prima parte; I validatori - seconda parte
- **Gestire l'applicazione**: Terminare la `wx.App`; Gestione degli *standard streams*; Logging - prima parte; Logging - seconda parte; Gestione delle eccezioni - prima parte; Gestione delle eccezioni - seconda parte; Localizzare un programma wxPython

3 Installazione e documentazione.

3.1 Che cosa è wxPython.

wxPython (https://www.wxpython.org) è un framework per la costruzione di *GUI* (programmi con interfaccia utente grafica) in Python. Si tratta di un *porting* di wxWidgets (https://wxwidgets.org), un framework C++ molto popolare e consolidato (la prima versione risale al 1992). Per la precisione, wxPython non è una riscrittura ma un *wrapper* di wxWidgets: ovvero un *layer* di codice Python che istanzia e manipola gli oggetti C++ sottostanti di wxWidgets. Non c'è bisogno di conoscere C++ (o peggio, di *scrivere* codice C++) per usare wxPython: tuttavia è utile sapere che wxPython «segue da vicino» wxWidgets e ne mappa fedelmente le *API*.

A sua volta wxWidgets si rivolge alle primitive grafiche della piattaforma ospitante per presentare all'utente un'interfaccia dall'aspetto «nativo». Questo vuol dire che, quando per esempio istanziamo un `wx.TextCtrl` per disegnare una normale casella di testo, wxPython delega il compito all'oggetto wxWidgets sottostante `wxTextCtrl` che a sua volta crea il *widget* nativo Windows, oppure quello GTK, oppure quello Cocoa, a seconda che il programma sia eseguito su Windows, Linux, MacOs.

Il vantaggio di questa impostazione è che l'utente vedrà la GUI del nostro programma con l'aspetto e le consuetudini a cui è abituato dal suo sistema operativo. Storicamente questo è stato un punto di forza per wxWidgets/wxPython soprattutto in ambiente Windows, dove è più sentita l'importanza di realizzare GUI dall'aspetto nativo. Lo svantaggio è che, per quanto wxWidgets faccia miracoli per presentare un «minimo comun denominatore» omogeneo tra le varie piattaforme, piccole differenze tra le diverse implementazioni si trovano sempre.

3.2 Phoenix e Classic.

Prima di vedere come è possibile ottenere e installare una versione «moderna» di wxPython, riassumiamo la situazione attuale dello sviluppo di questo framework.

La versione storica di wxPython è oggi chiamata «wxPython Classic», e corrisponde alle serie 2.x e 3.x (la differenza è minima e riguarda la versione di wxWidgets sottostante). wxPython Classic *non* supporta Python 3, e *non* è più attivamente sviluppato.

La serie attuale (4.x) viene chiamata «wxPython Phoenix» e supporta *sia* Python 2.7 *sia* Python 3 (dalla 3.4 in poi).

Oggi non ha più praticamente nessun senso installare ancora wxPython Classic: per i nuovi progetti è praticamente obbligatorio usare Phoenix, tanto più che Phoenix supporta *anche* Python 2.7. D'altra parte le vecchie applicazioni basate su wxPython Classic andrebbero migrate quanto prima a

Phoenix, *se non altro* perché dovrebbero migrare a Python 3 (che Classic non supporta). Ricordiamo infatti che Python 2 non sarà più supportato dalla Python Software Foundation a partire dalla fine del 2019. In pratica quindi ha ancora senso procurarsi una vecchia versione di wxPython Classic solo se occorre mantenere vecchie applicazioni *senza migrarle*, consapevoli però del fatto che saranno basate su software (Python 2 e wxPython Classic) non più aggiornati, *neppure per gli aspetti relativi alla sicurezza*. Inutile dire che sarebbe meglio non fare niente del genere.

3.3 Installazione.

Fino a qualche anno fa l'installazione di wxPython (Classic) era problematica: il pacchetto era troppo ingombrante per essere distribuito normalmente su PyPI (https://pypi.org/), la *repository* ufficiale delle librerie Python. Erano quindi disponibili diversi *installer* precompilati per le varie piattaforme, scaricabili dal sito di wxPython.

Oggi il limite di «peso» per i pacchetti è stato rimosso, e anche wxPython è finalmente disponibile su PyPI (https://pypi.org/project/wxPython/) e pertanto installabile con Pip come qualsiasi altro pacchetto Python:

```
pip install wxPython
```

Attualmente sono disponibili *wheel* precompilate per Mac e Windows, per Python 2.7 e 3.4+. È naturalmente possibile, e anzi consigliabile, installare wxPython in un *virtual environment* oppure, in futuro, includerlo nella directory `__pypackages__` di un progetto, quando la PEP 582 (https://www.python.org/dev/peps/pep-0582/) verrà accettata.

Dopo aver installato wxPython, conviene eseguire dalla shell la *utility* `wxdemo` che scarica la Demo di wxPython e la installa separatamente. Successivamente è possibile invocare `wxdemo` come un *launcher* della Demo così installata.

Analogamente, è possibile eseguire `wxdocs` per scaricare e poi aprire nel browser una versione *offline* della documentazione di wxPython.

Una risorsa alternativa per scaricare wxPython è il download diretto dal sito ufficiale (https://extras.wxpython.org/wxPython4/extras/): qui si possono trovare inoltre le versioni precedenti di wxPython Phoenix, la Demo, la documentazione e soprattutto le versioni precompilate per le più note distribuzioni Linux (non disponibili su PyPI).

Infine è possibile scaricare le *snapshot-builts* da una pagina separata del sito ufficiale (https://wxpython.org/Phoenix/snapshot-builds/): si tratta di *built* automatici che vengono compilati quando il codice di wxPython è modificato nella sua repository su GitHub (https://github.com/wxWidgets/Phoenix). Dal momento che le *release* ufficiali possono tardare, talvolta può essere utile installare invece una *snapshot-built* più recente, quando siamo certi che porta la risoluzione di un baco altrimenti bloccante per la nostra applicazione. Ovviamente queste *built* non sono da considerarsi definitive, ed è sempre meglio sostituirle con la *release* successiva non appena questa è disponibile.

---

wxClassic: Le vecchie versioni di wxPython Classic sono ancora disponibili su SourceForge (https://sourceforge.net/projects/wxpython/files/wxPython/), comprese la Demo e la documentazione.

3.4 Documentazione.

La documentazione di wxPython, che in passato era molto frammentaria e lacunosa, è migliorata decisamente con Phoenix: adesso tutte le *API* sono documentate in una interfaccia gradevole e di facile consultazione. La documentazione è disponibile online (https://wxpython.org/Phoenix/docs/html/) ma, come abbiamo visto, è più facile scaricarla con `wxdocs` e consultarla localmente.

Accanto alla documentazione delle API è utile avere anche la Demo: si tratta di un programma che raccoglie decine di esempi di ciò che si può fare con wxPython, organizzati in una interfaccia molto facile da sfogliare. Ogni pagina della Demo è una dimostrazione «live» di un *widget* o un aspetto della programmazione wxPython: si può ispezionare il codice di esempio, eventualmente modificarlo e vedere interattivamente il risultato.

Per quanto efficace, la Demo non è esente da difetti: non tutto è documentato; gli esempi sono stati scritti in periodi e da persone diverse e talvolta si vede che il codice usa strumenti e consuetudini che nel frattempo sono stati migliorati o rimpiazzati; talvolta il codice è iper-semplificato, talvolta è sintetico e oscuro nel tentativo di condensare in poche righe molte possibilità e *feature* differenti. In ogni caso la Demo resta sempre la prima fermata obbligatoria per esplorare le infinite possibilità di wxPython.

Infine, resta sempre disponibile la documentazione di wxWidgets (https://docs.wxwidgets.org), ovvero il *framework* C++ sottostante. Una volta era praticamente obbligatorio sapervisi orientare per colmare le lacune nella documentazione di wxPython; oggi può essere tranquillamente ignorata almeno per le esigenze più comuni.

Il Wiki di wxPython (https://wiki.wxpython.org/) è attualmente in uno stato molto frammentario: la quasi totalità delle pagine è ancora pensata per wxPython Classic, e anzi molte informazioni sono vecchie anche rispetto alle ultime *release* di Classic. Tuttavia, a sfogliarlo con pazienza, si possono trovare molte autentiche gemme. Come ultima risorsa è sempre possibile rivolgere una domanda sul forum dedicato wxPython-users (https://groups.google.com/forum/#!forum/wxpython-users) oppure wxPython-dev (https://groups.google.com/forum/#!forum/wxpython-dev).

23

Selettori: offrire una scelta tra più opzioni.

In questo capitolo passiamo in rassegna una serie di *widget* di uso comune che risolvono con approcci diversi lo stesso compito: presentare all'utente una ristretta lista di opzioni e permettergli di sceglierne una o più di una.

Il generale questi widget accettano un argomento `choices` del costruttore: una lista di stringhe che raccoglie le opzioni da presentare. Hanno quindi alcuni metodi *getter* e *setter* per lavorare con l'elemento selezionato (alcuni ammettono la selezione multipla), ed emettono degli eventi tipici quando l'utente cambia la selezione. Per alcuni widget è anche possibile modificare a *runtime* l'elenco delle opzioni. In questo capitolo descriviamo solo le caratteristiche più importanti: rimandiamo come sempre alla documentazione per approfondimenti.

Si tenga conto che questi widget sono concepiti per presentare liste semplici (una sola colonna) e possibilmente corte. Se c'è bisogno di strumenti più complessi, allora `wx.ListCtrl` è probabilmente la soluzione adatta.

23.1 `wx.RadioButton` e `wx.RadioBox`.

La classe `wx.RadioButton` rappresenta la più semplice delle opzioni possibili: un pulsante «radio» che può essere solo «On» oppure «Off». Il costruttore accetta un parametro opzionale `label` che associa un'etichetta esplicativa al pulsante.

Un `wx.RadioButton` da solo non serve a nulla, e resterà sempre nello stato «On»: quando occorre presentare una sola opzione, lo strumento giusto è `wx.CheckBox` come vedremo tra poco. È possibile invece raggruppare diversi `wx.RadioButton` per offrire all'utente una scelta (esclusiva) tra più opzioni. Per fare questo, passiamo lo stile `wx.RB_GROUP` al *primo* pulsante del gruppo, e nessuno stile ai pulsanti successivi. Il gruppo termina quando non ci sono più pulsanti, oppure quando un pulsante ha di nuovo lo stile `wx.RB_GROUP`, che indica l'inizio di un nuovo gruppo.

Le interfacce `wx.<Window>.Get|SetValue` accettano e restituiscono un valore booleano per indicare lo stato del pulsante. Possiamo usare il *setter* solo per impostare il valore a `True`: questo disattiverà contemporaneamente gli altri pulsanti del gruppo.

L'evento `wx.EVT_RADIOBUTTON` viene emesso quando l'utente agisce sul pulsante. Non è un problema collegare tutti i pulsanti del gruppo (o addirittura quelli di diversi gruppi) a un solo *callback*: l'azione dell'utente può avere solo l'effetto di attivare un pulsante, e pertanto ci basta recuperare il riferimento all'oggetto che ha emesso l'evento, e quindi la sua etichetta:

```
class MainFrame(wx.Frame):
    def __init__(self, *args, **kwargs):
        wx.Frame.__init__(self, *args, **kwargs)
        p = wx.Panel(self)
        r1 = wx.RadioButton(p, -1, '1', style=wx.RB_GROUP, pos=(10, 30))
        r2 = wx.RadioButton(p, -1, '2', pos=(10, 50))
        r3 = wx.RadioButton(p, -1, '3', pos=(10, 70))
        # inizia un secondo gruppo
        r4 = wx.RadioButton(p, -1, 'A', style=wx.RB_GROUP, pos=(10, 90))
        r5 = wx.RadioButton(p, -1, 'B', pos=(10, 110))
        r6 = wx.RadioButton(p, -1, 'C', pos=(10, 130))
        for rb in (r1, r2, r3, r4, r5, r6):
            rb.Bind(wx.EVT_RADIOBUTTON, self.on_radio)

    def on_radio(self, evt):
        print(evt.GetEventObject().GetLabel())
```

La classe `wx.RadioBox` semplifica il processo di creare gruppi di pulsanti: passiamo al parametro `choices` del costruttore una lista di opzioni, e wxPython crea per noi un pulsante per ciascuna di queste. L'unico svantaggio è che non possiamo decidere il layout esatto del gruppo. Possiamo dare solo alcune indicazioni generali attraverso gli stili:

- `wx.RA_HORIZONTAL` e `wx.RA_VERTICAL` allineano i pulsanti in riga oppure in colonna;
- se le opzioni sono troppo numerose e desideriamo raggrupparle in un formato tabellare, passiamo al costruttore anche il parametro `majorDimension` per indicare il numero massimo di righe o colonne che vogliamo, e lo stile `wx.RA_SPECIFY_ROWS` o `wx.RA_SPECIFY_COLS` per dire se `majorDimension` si riferisce alle righe o alle colonne.

Per esempio:

```
# raggruppa le opzioni in 2 righe, da 5 e 4 elementi ciascuna
rb = wx.RadioBox(parent, choices=list('123456789'),
                 majorDimension=2, style=wx.RA_SPECIFY_ROWS)
```

Usiamo `wx.RadioBox.GetSelection` per sapere l'indice dell'elemento selezionato, mentre `wx.RadioBox.GetString` ci restituisce l'etichetta:

```
selected = radio_box.GetString(radio_box.GetSelection())
```

L'evento caratteristico di un `wx.RadioBox` è `wx.EVT_RADIOBOX`.

23.2 `wx.CheckBox`.

Questa classe rappresenta la consueta «casella con la spunta» che può avere tipicamente due stati, «On» e «Off». In Windows è possibile creare una casella a tre stati (di solito usata per includere uno stato sconosciuto, non applicabile, etc.). Un `wx.CheckBox` è sempre isolato: non esiste il concetto di gruppo di opzioni come per i `wx.RadioButton`. Possiamo certamente allineare diversi `wx.CheckBox` e considerarli parte di un unico gruppo logico ai fini della nostra applicazione, ma wxPython li tratta comunque come elementi separati.

L'uso di un `wx.CheckBox` è molto semplice. Passiamo un parametro `label` al costruttore per associare alla casella un'etichetta esplicativa, ed eventualmente lo stile `wx.CHK_3STATE` per creare una casella a tre stati. Aggiungiamo lo stile `wx.CHK_ALLOW_3RD_STATE_FOR_USER` se vogliamo permettere all'utente di selezionare anche lo stato «incerto».

Le normali interfacce `wx.<Window>.Get|SetValue` funzionano solo per la casella a due stati. Occorre usare `wx.CheckBox.Get|Set3StateValue` per quella a tre stati. Infine, `wx.CheckBox.IsChecked` è un alias più leggibile per il *getter*.

L'evento tipico emesso è `wx.EVT_CHECKBOX`. Nel callback possiamo interrogare `wx.Event.IsChecked` per conoscere lo stato della casella che ha originato l'evento.

23.3 Widget con liste di opzioni.

Presentiamo adesso alcuni widget che elencano le opzioni come una lista cliccabile. Sono tutti presenti nella demo, alla quale rimandiamo per vederli in azione.

Questi widget derivano come gli altri da `wx.Control`, ma anche dalla classe *mixin* `wx.ItemContainer` che conferisce la capacità di presentare e gestire liste di valori, anche modificandole a *runtime*:

- `wx.ItemContainer.Append` aggiunge uno o più elementi in coda alla lista;
- `wx.ItemContainer.Set(items)` rimpiazza il contenuto di una lista con dei nuovi elementi;
- `wx.ItemContainer.Insert(item, pos)` inserisce un nuovo elemento nel punto voluto (ma possiamo passargli anche una lista di elementi);
- `wx.ItemContainer.Delete(n)` elimina lo n-esimo elemento;
- `wx.ItemContainer.Clear` svuota completamente la lista;
- `wx.ItemContainer.GetItems` restituisce gli elementi della lista.
- `wx.ItemContainer.GetSelection` restituiscono l'indice dell'elemento selezionato;
- `wx.ItemContainer.GetString(index)` permette di risalire alla stringa di testo;
- `wx.ItemContainer.SetSelection(index)` seleziona una voce.

23.3.1 `wx.ListBox`.

Il più semplice di questa famiglia di widget è `wx.ListBox`, che incolonna la lista in un semplice rettangolo: quando le opzioni diventano troppe, aggiunge una barra di scorrimento verticale. Oltre al consueto argomento `choices`, possiamo passare al costruttore una serie di stili, i più notevoli dei quali sono `wx.LB_MULTIPLE` e `wx.LB_EXTENDED` che creano una lista a selezione multipla (il primo stile permette di selezionare solo con il mouse, il secondo con mouse e tastiera).

Questo widget emette due eventi tipici: `wx.EVT_LISTBOX` quando l'utente fa clic su un elemento, e `wx.EVT_LISTBOX_DCLICK` quando fa doppio clic.

L'interfaccia è estremamente semplice: in pratica oltre agli strumenti ereditati da `wx.ItemContainer` bisogna considerare solo le interfacce `wx.ListBox.Get|SetSelections` che lavorano sulle liste a selezione multipla.

23.3.2 `wx.CheckListBox`.

Una versione più complessa dello stesso widget è `wx.CheckListBox`: in questo caso a ciascuna voce della lista è associata una casella cliccabile `wx.CheckBox`. Di conseguenza le voci si possono non solo selezionare, ma anche «spuntare». Questo widget è una sotto-classe di `wx.ListBox`, e quindi ne eredita stili e metodi. In teoria è quindi possibile creare un `wx.CheckListBox` a scelta multipla: in pratica però sarebbe solo disorientante per l'utente, e conviene limitarsi alla selezione singola. Un `wx.CheckListBox` a selezione singola equivale in sostanza a un `wx.ListBox` multiplo più sicuro: la

selezione avviene spuntando la casella, e l'utente non corre il rischio di perderla per un clic fuori posto.

L'interfaccia di `wx.CheckListBox` aggiunge alcuni metodi dedicati a gestire le voci spuntate: `wx.CheckListBox.Get|SetCheckedStrings` accettano e restituiscono un elenco di voci della lista (non indici, quindi); `wx.CheckListBox.Get|SetCheckedItems` fanno lo stesso ma lavorano con gli indici; `IsChecked(index)` controlla se una voce è spuntata. L'evento caratteristico emesso è `wx.EVT_CHECKLISTBOX`.

23.3.3 `wx.Choice`.

Questo widget è il classico elenco *drop-down* che siamo abituati a vedere ovunque: in pratica, è un `wx.ListBox` a scelta singola, che mostra solo la voce selezionata in quel momento.

L'interfaccia è identica a quella di un normale `wx.ListBox`, tranne che per l'impossibilità di renderlo a scelta multipla. Il widget emette un evento caratteristico `wx.EVT_CHOICE`

23.3.4 `wx.ComboBox`.

Questo widget combina insieme un `wx.Choice` e una casella di testo `wx.TextCtrl`: in pratica l'utente può selezionare una voce della lista, oppure inserire lui stesso una stringa di testo. L'aspetto del `wx.ComboBox` non è omogeneo per tutte le piattaforme: in Windows è possibile passare lo stile `wx.CB_SIMPLE` al costruttore per utilizzare un normale `wx.ListBox` invece della lista *drop-down*. Esiste anche uno stile `wx.CB_READONLY` che non permette all'utente di inserire delle voci nuove, forzando il widget a comportarsi in sostanza come un `wx.Choice` con una casella di testo. Rimandiamo alla documentazione per i dettagli delle varie implementazioni.

Un `wx.ComboBox` emette il suo evento caratteristico `wx.EVT_COMBOBOX` quando l'utente effettua una scelta, ma emette anche gli eventi di una casella di testo: `wx.EVT_TEXT` ed eventualmente `wx.EVT_TEXT_ENTER` (se abbiamo inizializzato il widget con lo stile `wx.TE_PROCESS_ENTER`).

L'interfaccia di `wx.ComboBox` è naturalmente simile a quella di un `wx.ListBox`, con l'aggiunta di metodi tipici dei `wx.TextCtrl` come `wx.<Window>.Get|SetValue`.

Nell'esempio che segue mostriamo un `wx.ComboBox` che aggiunge automaticamente valori alla lista quando l'utente dà `<invio>`. Anche per variare rispetto allo schema che abbiamo usato finora, scriviamo qui una sotto-classe specializzata di `wx.ComboBox` dove gestiamo la meccanica «interna» della lista, ovvero il meccanismo di popolamento che si innesca in seguito a un `wx.EVT_TEXT_ENTER`. Il comportamento da tenere in seguito a una selezione (`wx.EVT_COMBOBOX`), d'altra parte, è più appropriato che sia deciso all'interno della finestra che ospita il nostro widget, e quindi lo lasciamo fuori dalla sotto-classe.

```
class MyCombo(wx.ComboBox):
    def __init__(self, *args, **kwargs):
        kwargs['style'] = wx.CB_DROPDOWN|wx.TE_PROCESS_ENTER|wx.CB_SORT  ❶
        wx.ComboBox.__init__(self, *args, **kwargs)
        self.Bind(wx.EVT_TEXT_ENTER, self.on_enter)

    def on_enter(self, evt):
        txt = self.GetValue().strip()                                    ❷
        if txt and txt not in self.GetItems():
            self.Append(txt)
        self.SetValue('')                                                ❷

class MainFrame(wx.Frame):
    def __init__(self, *args, **kwargs):
```

(continua...)

(...segue)

```
        wx.Frame.__init__(self, *args, **kwargs)
        panel = wx.Panel(self)
        self.cb = MyCombo(panel, choices=['foo', 'bar', 'baz'],
                          pos=(10, 10), size=(150, -1))                    ❸
        self.cb.Bind(wx.EVT_COMBOBOX, self.on_combo)

    def on_combo(self, evt):
        print(self.cb.GetString(self.cb.GetSelection()))
```

Per prima cosa, una piccola ineleganza: nella nostra sotto-classe, per fare prima, forziamo il parametro `style` dietro le quinte (❶), mantenendo in apparenza la stessa *signature* di `wx.ComboBox`. Questo è cattivo design *OOP*, beninteso. Bisognerebbe scrivere una *API* specifica per `MyCombo`, e documentarla.

Per il resto, il codice si spiega quasi da sé. Il nostro widget ha lo stile `wx.TE_PROCESS_ENTER` (❶), senza il quale non potrebbe processare l'evento relativo al tasto `<invio>`. Gli diamo anche lo stile `wx.CB_SORT` per fargli ordinare automaticamente le nuove voci. Si noti anche l'utilizzo dei consueti metodi `wx.<Window>.Get|SetValue` (❷) che manipolano la casella di testo interna al `wx.ComboBox`. Infine, un dettaglio: la larghezza naturale di un `wx.ComboBox` è quella della sua voce interna più lunga. Abbiamo dovuto specificare il parametro `size` del costruttore (❸) per renderlo più maneggevole: 150 pixel di larghezza, altezza di default (`-1`). Naturalmente, se avessimo *usato i sizer* (pagina 103) invece del posizionamento assoluto, il problema si sarebbe risolto da solo.

23.3.5 Associare *client data* alle opzioni.

I widget «a lista» che abbiamo presentato fin qui sono concepiti per mostrare all'utente una serie di opzioni relativamente corta e statica. Se abbiamo bisogno di mostrare liste più complesse, `wx.ListCtrl` offre molta più flessibilità.

Tuttavia anche questi widget offrono un minimo supporto per la logica *MCV*: mantenere la *View* (ovvero, i dati presentati) separata da un *Model* sottostante (i dati veri). La classe `wx.ItemContainer`, da cui derivano, implementa il concetto di *client data* da associare agli elementi della lista. In pratica si tratta di un oggetto arbitrario che il widget non mostrerà mai all'utente, ma che può contenere qualsiasi ulteriore informazione sulla specifica voce di un elenco.

L'interfaccia per usare i *client data* è semplice, ed è comune a tutti i widget appena visti:

- `wx.ItemContainer.Append(item, clientData)` ha un secondo argomento opzionale `clietData` che ci consente di specificare i dati «nascosti» di un elemento, al momento di inserirlo nella lista;
- in modo analogo funziona `wx.ItemContainer.Insert(item, pos, clientData)`;
- `wx.ItemContainer.GetClientData(n)` recupera il *client data* dello n-esimo elemento;
- `wx.ItemContainer.SetClientData(n, data)` assegna un *client data* allo n-esimo elemento.

Possiamo usare i *client data* per conservare qualsiasi riferimento vogliamo al *Model* sottostante, se ne abbiamo uno: per esempio, l'id univoco di una riga di database. In questo modo possiamo disaccoppiare i dati veri dal modo, variabile, in cui i dati sono presentati.

Consideriamo per esempio un `wx.ListBox` che elenca una serie di nomi, che supponiamo siano estratti da un database: mentre noi presentiamo all'utente solo il nome, memorizziamo tuttavia l'Id della riga del database nel *client data* associato. In questo modo, quando l'utente seleziona un nome, noi recuperiamo facilmente l'Id e possiamo usarlo per ulteriori richieste al database (per esempio per vedere i «dettagli del contatto» etc.):

```
# per es. in seguito a una query "SELECT id, name FROM names..."
all_names = ((12, 'Giorgio'), (7, 'Paola'), (25, 'Giuseppe'), ...)

class MainFrame(wx.Frame):
    def __init__(self, *args, **kwargs):
        wx.Frame.__init__(self, *args, **kwargs)
        self.namelist = wx.ListBox(self)
        for id, name in all_names:
            self.namelist.Append(name, id)
        self.namelist.Bind(wx.EVT_LISTBOX, self.on_select)

    def on_select(self, event):
        id = self.namelist.GetClientData(event.GetSelection())
        # o direttamente: id = event.GetClientData()
```

23.4 wx.Slider.

Questa classe rappresenta un comune *slider*, e offre all'utente la scelta tra un *range* di valori numerici consecutivi. Il suo utilizzo è molto semplice: il costruttore accetta gli argomenti `minValue` e `maxValue` per determinare l'intervallo di azione, e `value` per impostare il valore iniziale. Le comuni operazioni `wx.<Window>.Get|SetValue`, che accettano e restituiscono numeri interi possono essere utilizzate per leggere e impostare il valore del widget. Per esempio,

```
slider = wx.Slider(parent, minValue=-10, maxValue=10, value=0,
                   style=wx.SL_HORIZONTAL)
```

disegna uno *slider* orizzontale con valori compresi tra 10 e -10.

Gli stili disponibili sono diversi, e invitiamo a leggere l'elenco completo nella documentazione. I due più comuni sono `wx.SL_HORIZONTAL` e `wx.SL_VERTICAL` che determinano l'orientamento dello *slider*. L'evento caratteristico è `wx.EVT_SLIDER`, che viene emesso ogni volta il valore del widget cambia.

23.5 wx.SpinButton e wx.SpinCtrl.

Concludiamo questo capitolo con due widget che, a differenza dei precedenti, non presentano una lista di opzioni ma permettono di selezionare un numero intero compreso in un intervallo. In questo sono più simili a `wx.Slider`, con la differenza che occupano meno spazio e sono più adatti quando il *range* è molto grande.

La classe `wx.SpinButton` rappresenta la versione più semplice, e disegna semplicemente due piccoli pulsanti a forma di freccia, senza nessuna indicazione visibile del valore numerico attuale. È adatto in pratica solo come elemento di un layout più articolato, dove altri widget si incaricano di restituire all'utente un effetto visibile della sua azione sui pulsanti.

Un `wx.SpinCtrl` aggiunge alle due frecce un `wx.TextCtrl` «sincronizzato» per visualizzare il valore. L'utente può anche immettere un numero direttamente nella casella di testo: i valori non validi sono respinti automaticamente. Nella pratica, un `wx.SpinCtrl` viene spesso usato con la funzione di una normale casella di testo per valori numerici: anche se i pulsanti sono piuttosto scomodi e l'utente non li usa quasi mai, il vantaggio è comunque che `wx.SpinCtrl` offre una pre-validazione dei valori immessi. Non è possibile inserire qualcosa di diverso da un numero, e bisogna rispettare l'intervallo prefissato. Naturalmente wxPython dispone di strumenti più raffinati (come i *masked controls*) per questi compiti: ma un `wx.SpinCtrl` è facile da usare.

In modo simile a uno *slider* il costruttore di `wx.SpinCtrl` accetta i parametri `min`, `max` e `initial` per determinare l'intervallo e il valore iniziale. Si noti però che il costruttore di `wx.SpinButton` non accetta questi parametri: è comunque possibile, per entrambi i widget, usare le interfacce `wx.<Window>.SetMax`, `wx.<Window>.SetMin` e `wx.<Window>.SetValue` per controllare l'intervallo di azione. Se non impostiamo nessun limite, un `wx.SpinButton` è libero di fluttuare lungo il *range* degli interi con segno disponibile sulla piattaforma.

Un `wx.SpinButton` dispone di alcuni stili, tra cui `wx.SP_HORIZONTAL` e `wx.SP_VERTICAL` per determinarne l'orientamento. Il `wx.SpinCtrl` supporta anche alcuni stili di `wx.TextCtrl`, tra cui `wx.TE_PROCESS_ENTER` che gli permette di essere sensibile alla pressione del tasto `<invio>`.

Gli eventi caratteristici emessi sono `wx.EVT_SPIN` e `wx.EVT_SPINCTRL`.

43 I validatori, prima parte.

La validazione dei dati, ovvero garantire che i dati immessi dall'utente siano conformi alle regole del dominio applicativo del nostro programma, è un processo delicato: wxPython ci viene incontro con uno strumento apposito, la classe `wx.Validator` che ha due funzionalità complementari:

- convalidare i dati;
- trasferire i dati dentro e fuori da una finestra di dialogo.

Possiamo usare i validatori anche solo per una di queste due funzioni o per entrambe, a nostro piacere. Dedichiamo questo capitolo a spiegare la funzione di validazione, e il successivo al trasferimento dei dati.

---

wxClassic: In wxPython Classic la classe da usare era `wx.PyValidator`, che in Phoenix è ormai deprecata insieme alle molte classi «Py» che un tempo offrivano funzionalità in più.

43.1 Come scrivere un validatore.

Occorre semplicemente sotto-classare `wx.Validator`. Ecco un esempio da manuale: questo è un validatore che si può applicare a una casella di testo, e che garantisce che l'utente non la lasci vuota:

```
class NotEmptyValidator(wx.Validator):
    def Clone(self): return NotEmptyValidator()                    ❶

    def Validate(self, ctl):
        win = self.GetWindow()                                     ❹
        val = win.GetValue().strip()
        if val == '':
            return False                                           ❷
        else:
            return True                                            ❸
```

La prima riga è *boilerplate* necessario (❶). `wx.Validator.Clone` deve esserci necessariamente, e deve restituire una istanza dello stesso validatore. La parte interessante è `wx.Validator.Validate`: sovrascriviamo questo metodo per fare la nostra validazione: dobbiamo restituire `True` (❷) se la validazione ha successo, `False` (❸) altrimenti. Dall'interno del validatore è possibile risalire a un'istanza del *widget* che stiamo validando, chiamando `wx.Validator.GetWindow` (❹). In effetti il secondo, necessario, parametro di `wx.Validator.Validate` punta al *parent* del widget associato.

Il costruttore di un validatore, di norma, non vuole nessun parametro. Tuttavia niente impedisce di passare parametri extra alle sotto-classi. Per esempio, questo validatore garantisce che il valore immesso non sia in una bad-list di parole proibite:

```
class NotInBadListValidator(wx.Validator):
    def __init__(self, badlist):
        wx.Validator.__init__(self)
        self._badlist=badlist

    def Clone(self): return NotInBadListValidator(self._badlist)          ❶

    def Validate(self, ctl):
        win = self.GetWindow()
        val = win.GetValue().strip()
        return val not in self._badlist
```

Non dimentichiamoci di riportare i parametri correttamente anche in `wx.Validator.Clone` (❶): anche il *boiledplate* richiede qualche attenzione.

43.1.1 Validazione e validazione a cascata.

Una volta scritto, il validatore si applica al widget che intendiamo validare, al momento della sua creazione, passandolo direttamente al costruttore (come parametro `validator`). Ovviamente possiamo usare diverse istanze dello stesso validatore per validare più widget. Ecco un esempio:

```
class YourNamePanel(wx.Panel):
    def __init__(self, *args, **kwargs):
        wx.Panel.__init__(self, *args, **kwargs)
        self.first_name = wx.TextCtrl(self,
                                      validator=NotEmptyValidator())      ❶
        self.family_name = wx.TextCtrl(self,
                                       validator=NotEmptyValidator())     ❷

        s = wx.FlexGridSizer(2, 2, 5, 5)
        s.AddGrowableCol(1)
        s.Add(wx.StaticText(self, -1, 'nome:'),
              0, wx.ALIGN_CENTER_VERTICAL)
        s.Add(self.first_name, 1, wx.EXPAND)
        s.Add(wx.StaticText(self, -1, 'cognome:'),
              0, wx.ALIGN_CENTER_VERTICAL)
        s.Add(self.family_name, 1, wx.EXPAND)
        self.SetSizer(s)
        s.Fit(self)

class MainFrame(wx.Frame):
    def __init__(self, *args, **kwargs):
        wx.Frame.__init__(self, *args, **kwargs)
        p = wx.Panel(self)
        self.your_name = YourNamePanel(p)
        validate = wx.Button(p, -1, 'valida')
        validate.Bind(wx.EVT_BUTTON, self.on_validate)

        s = wx.BoxSizer(wx.VERTICAL)
        s.Add(self.your_name, 1, wx.EXPAND)
        s.Add(validate, 0, wx.EXPAND|wx.ALL, 5)
        p.SetSizer(s)
```

(continua...)

(...segue)

```
    def on_validate(self, event):
        ret = self.your_name.Validate()
        if ret == False:
            wx.MessageBox('Non valido')
```

Come si vede (❶ e ❷), due caselle di testo sono collegate al nostro validatore. Possiamo testare anche l'altro che abbiamo scritto: basta cambiare la riga ❶ con qualcosa come:

```
invalid_names = ('Foo', 'Bar', 'Baz')
self.first_name = wx.TextCtrl(self,
                              validator=NotInBadListValidator(invalid_names))
```

Una volta che un widget ha un validatore assegnato, è possibile validarne il contenuto chiamando manualmente `wx.Window.Validate` su di esso. Questo metodo chiama a sua volta `wx.Validator.Validate` sul validatore associato al widget, che esegue la validazione. Tuttavia ci sono delle scorciatoie possibili. Notiamo infatti che nel nostro esempio abbiamo incorporato le caselle di testo in un *panel*: questo in parte perché è buona pratica raggruppare le funzionalità della *GUI* in piccoli «mattoni» coerenti, *come abbiamo già detto* (pagina 76). In questo caso specifico però il panel ci torna utile soprattutto per dimostrare la validazione «a cascata»: quando chiamiamo `wx.Window.Validate` sul panel, in effetti vengono validati tutti i widget figli del panel (purché abbiano un validatore associato, naturalmente). `wx.Window.Validate` chiamato sul panel restituisce `True` solo se tutti i figli passano la validazione, `False` altrimenti.

Possiamo assegnare un validatore a un widget anche dopo che è stato creato, chiamando `wx.Window.SetValidator`. Questo talvolta si rende necessario perché alcuni widget *non* prevedono il parametro `validator` nel costruttore (occorre sempre controllare la documentazione in merito). Se chiamiamo `wx.Window.SetValidator` su un widget che ha già un validatore, ogni volta l'ultimo sostituisce il precedente.

43.1.2 Quando fallisce una validazione a cascata.

Nel caso di validazione a cascata, abbiamo però un problema aggiuntivo: il processo di validazione si ferma non appena uno dei test fallisce, ma il valore restituito `False` non ci dice nulla di quale widget esattamente non ha superato la validazione.

Quando è necessario dare all'utente questa informazione, occorre far sì che sia il validatore stesso a occuparsene, invece del codice chiamante (che riceve solo il `False` di ritorno). Per esempio, possiamo riscrivere il nostro `NotEmptyValidator` in questo modo:

```
class NotEmptyValidator(wx.Validator):
    def Clone(self): return NotEmptyValidator()

    def Validate(self, ctl):
        win = self.GetWindow()
        val = win.GetValue().strip()
        if val == '':
            wx.MessageBox('Bisogna inserire del testo')
            return False
        else:
            return True
```

Questo però non è ancora sufficiente: se più caselle di testo hanno lo stesso validatore, talvolta si vuole sapere esattamente quale non funziona. Possiamo fare in molti modi, per esempio modificando anche il colore del widget incriminato:

```
class NotEmptyValidator(wx.Validator):
    def Clone(self): return NotEmptyValidator()

    def Validate(self, ctl):
        win = self.GetWindow()
        val = win.GetValue().strip()
        if val == '':
            win.SetBackgroundColour(wx.YELLOW)
            win.Refresh() # necessario!
            wx.MessageBox('Bisogna inserire del testo')
            return False
        else:
            # assicuriamoci di impostare il colore normale
            col = wx.SystemSettings.GetColour(wx.SYS_COLOUR_WINDOW)
            win.SetBackgroundColour(col)
            win.Refresh()
            return True
```

Un'altra soluzione potrebbe essere recuperare il *name* del widget, con una tecnica *che già conosciamo* (pagina 91):

```
class NotEmptyValidator(wx.Validator):
    def Clone(self): return NotEmptyValidator()

    def Validate(self, ctl):
        win = self.GetWindow()
        val = win.GetValue().strip()
        if val == '':
            msg = '%s: manca del testo!' % win.GetName()
            wx.MessageBox(msg)
            return False
        else:
            return True
```

Naturalmente questo sistema funziona solo se noi abbiamo in precedenza assegnato un parametro `name` significativo a ogni widget a cui associamo il validatore. Nel nostro esempio sarebbe:

```
self.first_name = wx.TextCtrl(self, name='Nome',
                              validator=NotEmptyValidator())
self.family_name = wx.TextCtrl(self, name='Cognome',
                               validator=NotEmptyValidator())
```

Il parametro `name` del costruttore di un widget non è di solito molto utile. In Python si possono passare gli oggetti stessi come parametri, e questo rende superfluo contrassegnare ciascun widget con un identificativo statico da passare in giro tra le varie funzioni; lo stesso discorso vale *per gli Id* (pagina 65). Questo però è uno dei casi in cui invece le *API* `wx.Window.Get|SetName` possono tornare utili per aggiungere un «nickname» piacevole al widget da presentare all'utente in caso di necessità.

43.1.3 La validazione ricorsiva.

La validazione a cascata si limita ai soli figli diretti, ma è possibile fare in modo che venga applicata ricorsivamente anche ai figli dei figli, e così via. Per fare questo occorre settare lo stile `wx.WX_EX_VALIDATE_RECURSIVELY`. Questo *è un extra-style* (pagina 71), e quindi va settato dopo la creazione, usando il metodo `wx.Window.SetExtraStyle`.

Facciamo degli esperimenti con il codice che abbiamo già scritto: per prima cosa, invece di validare il panel, proviamo a validare direttamente il frame:

```
class MainFrame(wx.Frame):
    # ...

    def on_validate(self, event):
        ret = self.Validate()  # era: ret = self.your_name.Validate()
        if ret == False:
            wx.MessageBox('Non valido')
```

Come previsto, la validazione non avviene. La *catena dei parent* (pagina 62) in effetti è lunga: dopo il frame c'è il panel contenitore (quello che nel nostro codice chiamiamo `p`), quindi l'istanza di `YourNamePanel`, e finalmente le caselle di testo che vogliamo validare.

Tuttavia, proviamo adesso ad aggiungere l'extra-style:

```
class MainFrame(wx.Frame):
    def __init__(self, *args, **kwargs):
        wx.Frame.__init__(self, *args, **kwargs)
        self.SetExtraStyle(wx.WS_EX_VALIDATE_RECURSIVELY)
        # ...
```

Ecco che la validazione avviene di nuovo.

43.2 La validazione automatica dei dialoghi.

Fin qui ci siamo limitati a chiamare `wx.Window.Validate` manualmente per effettuare la validazione. L'unico automatismo possibile è che, chiamandolo su un panel, si possono validare a cascata tutti i figli diretti (ed eventualmente anche i nipoti etc., usando la validazione ricorsiva).

Le *finestre di dialogo* (pagina 76), tuttavia, hanno una marcia in più. È possibile validare automaticamente un dialogo, quando è dotato di un pulsante con *Id predefinito* (pagina 67) `wx.ID_OK`. In questo caso, quando l'utente fa clic sul pulsante `wx.ID_OK`, il dialogo chiama automaticamente `wx.Window.Validate` su se stesso, prima di chiudersi. Se i widget contenuti nel dialogo hanno dei validatori assegnati, questi entreranno in funzione.

Riscriviamo l'esempio che abbiamo seguito finora, trasportato in un dialogo con validazione automatica:

```
class NotEmptyValidator(wx.Validator):
    def Clone(self): return NotEmptyValidator()
    def TransferToWindow(self): return True                    ❶
    def TransferFromWindow(self): return True                  ❶

    def Validate(self, ctl):
        win = self.GetWindow()
        val = win.GetValue().strip()
        if val == '':
```

(continua...)

(...segue)

```
            msg = '%s: manca del testo!' % win.GetName()
            wx.MessageBox(msg)
            return False
        else:
            return True

class NameDialog(wx.Dialog):
    def __init__(self, *args, **kwargs):
        wx.Dialog.__init__(self, *args, **kwargs)
        self.first_name = wx.TextCtrl(self, name='Nome',
                                      validator=NotEmptyValidator())
        self.family_name = wx.TextCtrl(self, name='Cognome',
                                       validator=NotEmptyValidator())
        validate = wx.Button(self, wx.ID_OK, 'valida')                    ❷
        cancel = wx.Button(self, wx.ID_CANCEL, 'annulla')                 ❷

        s = wx.FlexGridSizer(2, 2, 5, 5)
        s.AddGrowableCol(1)
        s.Add(wx.StaticText(self, -1, 'nome:'),
              0, wx.ALIGN_CENTER_VERTICAL)
        s.Add(self.first_name, 1, wx.EXPAND)
        s.Add(wx.StaticText(self, -1, 'cognome:'),
              0, wx.ALIGN_CENTER_VERTICAL)
        s.Add(self.family_name, 1, wx.EXPAND)

        s1 = wx.BoxSizer()
        s1.Add(validate, 1, wx.EXPAND|wx.ALL, 5)
        s1.Add(cancel, 1, wx.EXPAND|wx.ALL, 5)

        s2 = wx.BoxSizer(wx.VERTICAL)
        s2.Add(s, 1, wx.EXPAND|wx.ALL, 5)
        s2.Add(s1, 0, wx.EXPAND)
        self.SetSizer(s2)
        s2.Fit(self)

class MainFrame(wx.Frame):
    def __init__(self, *args, **kwargs):
        wx.Frame.__init__(self, *args, **kwargs)
        b = wx.Button(self, -1, 'mostra dialogo', pos=(10, 10))
        b.Bind(wx.EVT_BUTTON, self.on_clic)

    def on_clic(self, event):
        with NameDialog(self) as dlg:
            if dlg.ShowModal() == wx.ID_OK:                               ❸
                print('clic su OK, validazione automatica')
            else:                                                         ❹
                print('clic su Annulla, nessuna validazione')
```

In primo luogo, notiamo che purtroppo il *boilerplate* necessario per il validatore è aumentato un po": adesso occorre anche sovrascrivere «a vuoto» i metodi `wx.Validator.TransferFrom|ToWindow` (❶) di cui non abbiamo nessun bisogno al momento, visto che servono per il trasferimento dei dati (lo vedremo nel prossimo capitolo). Purtroppo però quando si tratta di validare una finestra di dialogo, wxPython se li aspetta in ogni caso, e si lamenta se non li trova. Il panel `YourNamePanel` è andato via, e i widget da validare sono stati inseriti direttamente nel dialogo `NameDialog`: sappiamo infatti che un `wx.Dialog` *ha già un suo panel incorporato* (pagina 76). Questo è un dettaglio importante:

ricordiamo infatti che la validazione automatica avverrà «a cascata» partendo *dalla finestra di dialogo* (ovvero chiamando `wx.Window.Validate` su questa), e quindi si arresterà ai widget del panel incorporato. Se il nostro layout prevede dei sotto-panel che contengono i widget da validare, ricordiamo di settare l'extra-style `wx.WX_EX_VALIDATE_RECURSIVELY` sulla finestra, per raggiungerli con la validazione ricorsiva. Abbiamo aggiunto infine un frame *top-level* `MainFrame` solo per esemplificare il modo di chiamare il dialogo e distruggerlo (automaticamente grazie all'uso del *context manager*).

Fatte queste premesse, passiamo alle cose più interessanti. Come abbiamo già visto *parlando degli Id* (pagina 67), i due pulsanti «Ok» e «Annulla» (❷) sanno già che cosa fare, senza bisogno di collegarli a un evento. Entrambi tentano di chiudere il dialogo, ma quello contrassegnato con `wx.ID_OK`, prima, esegue la validazione automatica. Tutti i widget nel dialogo vengono validati e *se la validazione fallisce il dialogo non si chiude*. Questo vuol dire che, finché la validazione non ha successo o l'utente non preme «Annulla», il codice chiamante (il *callback* `on_clic`) resta bloccato. Ecco un esempio chiaro in cui non c'è modo di affidare *al codice chiamante* il compito di informare l'utente sul risultato della validazione: *sono i validatori stessi* a doverlo fare, e che quindi devono avere il codice necessario per questo compito.

Il codice chiamante prosegue la sua corsa quando la validazione ha successo, il dialogo si chiude e `wx.Dialog.ShowModal` restituisce il codice corrispondente al pulsante premuto. Se adesso il codice è `wx.ID_OK` (❸), si può stare sicuri che i dati sono validi. Se il codice è `wx.ID_CANCEL` (❹), allora la validazione *non è avvenuta* e i dati non sono sicuri: in questo caso è ovviamente inutile recuperarli dal dialogo. Questo è un dettaglio importante: la validazione avviene solo in caso di `wx.ID_OK`. Se si desidera che i widget siano validati sempre, qualunque pulsante sia stato premuto, allora bisogna tornare alla validazione manuale: collegare i pulsanti a un evento, e chiamare `wx.Window.Validate` nel callback relativo.

43.3 Validazione e catena degli eventi.

Abbiamo fatto cenno di sfuggita *nel capitolo sugli handler* (pagina 169) che la validazione interviene in un momento ben preciso della linea di propagazione dell'evento: il comportamento di default che wxPython produce nel metodo `wx.EvtHandler.TryAfter`. È possibile inserire un handler personalizzato nella catena degli eventi e sovrascrivere `wx.EvtHandler.TryAfter` per intervenire al momento preciso in cui avviene la validazione. Ecco un esempio minimo che si limita a tener traccia dei vari passaggi:

```
class NotEmptyValidator(wx.Validator):
    def Clone(self): return NotEmptyValidator()
    def TransferToWindow(self): return True
    def TransferFromWindow(self): return True

    def Validate(self, ctl):
        win = self.GetWindow()
        val = win.GetValue().strip()
        if val == '':
            print('nel validatore: validazione fallita')
            return False
        else:
            print('nel validatore: validazione riuscita')
            return True

class MyHandler(wx.EvtHandler):
    def TryAfter(self, evt):
        if evt.GetEventType() == wx.wxEVT_BUTTON:
            print('in EvtHandler.TryAfter, prima del gestore wxPython')
```

(continua...)

(...segue)

```
            # qui avviene la validazione
            ret =  wx.EvtHandler.TryAfter(self, evt)
            print('in EvtHandler.TryAfter, dopo il gestore wxPython')
            return ret
        else:
            return wx.EvtHandler.TryAfter(self, evt)

class TestDialog(wx.Dialog):
    def __init__(self, *args, **kwargs):
        wx.Dialog.__init__(self, *args, **kwargs)
        self.txt = wx.TextCtrl(self, pos=(10, 10),
                               validator=NotEmptyValidator())
        self.validate = wx.Button(self, wx.ID_OK, 'valida', pos=(10, 50))
        self.validate.PushEventHandler(MyHandler())
        self.Bind(wx.EVT_WINDOW_DESTROY, self.on_destroy)

    def on_destroy(self, event):
        self.validate.PopEventHandler()
        event.Skip()

class MainFrame(wx.Frame):
    def __init__(self, *args, **kwargs):
        wx.Frame.__init__(self, *args, **kwargs)
        b = wx.Button(self, -1, 'mostra dialogo', pos=(10, 10))
        b.Bind(wx.EVT_BUTTON, self.on_clic)

    def on_clic(self, event):
        dlg = TestDialog(self)
        dlg.ShowModal()
        dlg.Destroy()
```

43.4 Consigli sulla validazione.

Terminiamo questo primo capitolo sui validatori con alcune riflessioni e consigli sulle tecniche più comuni.

43.4.1 Composizione di validatori.

A prima vista i validatori sembrano oggetti limitati: per esempio, non possono essere combinati tra loro per eseguire diversi test su un unico widget. Non è possibile chiamare diversi validatori uno dopo l'altro sullo stesso widget. Così ogni validatore dovrebbe avere nel suo metodo `wx.Validator.Validate` tutti i test che servono per un dato widget in una data circostanza. Questo limita il riutilizzo del validatore per diversi widget in condizioni differenti.

In realtà questa «limitazione» dipende di solito da un utilizzo errato dei validatori. L'errore è pensare che questi siano il posto in cui scrivere effettivamente i test di validazione. Dovrebbero essere invece solo il punto di raccordo finale tra la nostra suite di test di validazione e il widget che dobbiamo validare. Il codice effettivamente contenuto in `wx.Validator.Validate` dovrebbe essere breve e avere solo quanto basta a gestire i dati in partenza e le risposte in arrivo. Per esempio qualcosa come:

```
def Validate(self, ctl):
    val = self.GetWindow().GetValue()
```

(continua...)

(...segue)

```
    if all([test_1(val), test_2(val), test_3(val)]):
        return True
    else:
        # informo l'utente che la validazione è fallita
        return False
```

Così è possibile scrivere separatamente i vari `test_1`, `test_2`, etc. in modo «atomico» e generale, e poi combinarli tra loro nel validatore a seconda dei casi (anche l'ordine di esecuzione si può naturalmente controllare). Nella peggiore delle ipotesi si dovrà scrivere un breve validatore per ciascun widget da validare: ogni validatore rappresenta una catena di test da eseguire.

In teoria si può andare anche oltre e scrivere un validatore *general purpose* con un numero variabile di test passati come parametri:

```
class GroupTestValidator(wx.Validator):
    def __init__(self, *tests):
        wx.Validator.__init__(self)
        self._tests = tests

    def Clone(self): return GroupTestValidator(self._tests)

    def Validate(self, ctl):
        val = self.GetWindow().GetValue()
        if all([test(val) for test in self._tests]):
            return True
        else:
            return False
```

che poi può essere assegnato a diversi widget con diversi parametri:

```
text_1 = wx.TextCtrl(..., validator=GroupTestValidator(test_1, test_2))
text_2 = wx.TextCtrl(..., validator=GroupTestValidator(test_1, test_3, test_4))
```

Naturalmente non bisogna esagerare: un singolo validatore «dinamico» non può certo bastare per tutte le esigenze della nostra applicazione.

43.4.2 Validazione a cascata.

Sulla validazione a cascata, bisogna dire che da un lato è comoda, dall'altro introduce dei limiti. Prima di tutto, la validazione si ferma al primo widget che fallisce, ma ciò impedisce all'utente di sapere se ci sono altri errori dopo il primo. È frustrante correggere un errore, premere di nuovo «invio», e scoprire che c'era un errore anche nel campo successivo. Se vogliamo che tutti i widget siano validati comunque, non c'è niente da fare: occorre rinunciare alla validazione a cascata e validare a mano tutti i widget. Fortunatamente in Python tutto diventa più semplice:

```
failed = []
for widget in (self.nome, self.cognome):
    if not widget.Validate():
        failed.append(widget)
# poi presento la lista degli errori, etc. etc.
```

43.4.3 Validazione a seconda del contesto.

Un altro limite dei validatori è che sono concepiti principalmente per validare un widget senza tenere conto del contesto (per esempio, del valore di altri widget). Beninteso, il «contesto» può essere

calcolato e passato al validatore come argomento aggiuntivo:

```
class ContextValidator(wx.Validator):
    def Clone(self): return ContextValidator()

    def Validate(self, ctl, context):
        val = self.GetWindow().GetValue()
        if all([test_1(val, context), test_2(val, context), ...]):
            ...

# e quindi, nel codice chiamante:
context = something() # per esempio, il valore di un altro widget
retcode = widget.Validate(context)
```

Questo ovviamente rende impossibile ogni tipo di validazione automatica, ma abbiamo visto che con Python in genere non è un problema.

Ma c'è di più: sempre grazie alla flessibilità di Python, possiamo anche far calcolare il contesto dinamicamente al validatore stesso. Possiamo spingerci a cose non proprio ortodosse come questo esempio, dove un validatore ammette che una casella di testo sia vuota solo se un'altra è piena:

```
class AlternateEmptyValidator(wx.Validator):
    def __init__(self, context_getter):
        wx.Validator.__init__(self)
        # un callable per ottenere il contesto
        self.context_getter = context_getter

    def Clone(self): return AlternateEmptyValidator(self.context_getter)

    def Validate(self, ctl):
        val = self.GetWindow().GetValue()
        context_val = self.context_getter()
        if context_val == val == '':
            return False
        return True
```

Questo validatore accetta come parametro un *callable* da chiamare all'occorrenza per ottenere il contesto di cui ha bisogno. Nel nostro caso, per esempio, il *callable* sarà il metodo `GetValue` di un altro widget. Occorre avere l'accortezza di *non usarlo così*, naturalmente:

```
text_1 = wx.TextCtrl(..., validator=AlternateEmptyValidator(text_2.GetValue))
text_2 = wx.TextCtrl(..., validator=AlternateEmptyValidator(text_1.GetValue))
```

perché al momento di assegnare il validatore a `text_1`, `text_2` non esiste ancora, e quindi neppure `text_2.GetValue`! Tuttavia, può essere usato senza problemi in questo modo:

```
text_1 = wx.TextCtrl(...)
text_2 = wx.TextCtrl(...)
text_1.SetValidator(AlternateEmptyValidator(text_2.GetValue))
text_2.SetValidator(AlternateEmptyValidator(text_1.GetValue))
```

La cosa importante è che, grazie a Python, passiamo direttamente il *callable* `GetValue` come argomento del validatore, lasciando a questi il compito di chiamarlo quando necessario.

43.4.4 Problemi con i *masked controls*.

I validatori non giocano bene con i *masked controls*: una famiglia di widget disponibili nel *sub-package* `wx.lib.masked`, dotati di un loro sistema di validazione interno, separato. Quando un *masked control* non è valido, produce un suo comportamento di default (per esempio si colora di giallo): ma siccome non ha un validatore vero e proprio attaccato, è difficile integrare questo suo comportamento in un processo di validazione a cascata, per esempio.

43.4.5 Problemi con i widget limitati.

Una situazione analoga è quella che capita con i numerosi widget che, in wxPython, hanno la possibilità di limitare automaticamente i valori immessi. Per esempio, un `wx.SpinCtrl` può impostare un massimo e un minimo. Un `wx.ListBox` o un `wx.ComboBox` si caricano con una lista di valori tra cui scegliere, e così via. In questi casi la validazione, in un certo senso, è preventiva: l'utente non può che inserire dati validi.

Non è detto che i validatori siano completamente fuori gioco in questo caso. Possiamo lasciare che sia un validatore a caricare i dati in un `wx.ComboBox`, o impostare i limiti di un `wx.SpinCtrl`: è la funzione di trasferimento dati che vedremo nel prossimo capitolo. In ogni caso, non è sempre facile gestire con disinvoltura questo doppio canale di validazione, per cui certi widget sono controllati «a priori» e altri «a posteriori».

43.4.6 Validazione ricorsiva.

Ancora qualche parola sulla validazione ricorsiva. In linea di principio sarebbe meglio non esagerare, specialmente se applicata alle finestre *top-level* che raggruppano (in vari panel) diverse macro-aree della nostra applicazione. Quando chiamiamo `wx.Window.Validate` sulla finestra perché vogliamo validare un certo settore, contemporaneamente validiamo anche tutti gli altri. Nella migliore delle ipotesi è una perdita di tempo; nella peggiore è un guaio, se in quel momento gli altri settori sono in uno stato provvisoriamente inconsistente.

La cosa migliore è affidarsi al principio «ogni area, un panel» e *validare i singoli panel*, facendo affidamento sul fatto che i loro figli diretti saranno i widget che davvero ci serve validare. Occasionalmente, quando uno di questi panel-area ha una gerarchia più complessa (contiene altri panel, che contengono i widget), allora possiamo settare `wx.WX_EX_VALIDATE_RECURSIVELY` solo per questi.

43.4.7 In conclusione: usare i validatori?

I validatori sono strumenti utili, ma può essere difficile farli funzionare in modo armonico. Da un lato, la loro praticità risalta soprattutto quando sono accoppiati alle finestre di dialogo, con il meccanismo della validazione a cascata e automatica. Basta fare clic su `wx.ID_OK`, e si ottiene gratis la validazione di tutto quanto (e il trasferimento dei dati da e verso il dialogo, come vedremo). D'altra parte però con un ciclo `for` in Python, anche la validazione manuale è molto agevole e consente inoltre di personalizzare più accuratamente che cosa e quando validare. Inoltre, sempre grazie a Python, è possibile scrivere validatori più generali e dinamici.

Anche dopo aver imparato a usare bene i validatori, potrebbe restare comunque un vago *code smell*, come si dice in gergo. Il fatto è che i validatori sono uno dei componenti di wxPython che operano nella zona grigia che, nella terminologia *MCV*, sta tra la *View* e il *Controller* della nostra applicazione. Il disagio è anzi destinato ad aumentare quando, nel prossimo capitolo, useremo i validatori per trasferire dati tra la *View* e il *Model*. Così wxPython, che idealmente dovrebbe servire a scrivere solo il codice della *View*, in realtà sconfina in territori che forse avremmo voluto tenere più separati. D'altra parte non è l'unico componente «di confine» che wxPython mette a disposizione: la scelta se usarli resta nelle nostre mani.

In ultima analisi, i validatori sono uno strumento che wxPython ci propone per affrontare un problema. Vanno senz'altro bene per i casi più semplici, e possono essere usati con successo anche in scenari più difficili: ma se non riusciamo ad armonizzarli nel nostro framework di validazione complessivo, possiamo tranquillamente rinunciarvi.

49 Gestione delle eccezioni - prima parte.

Affrontiamo in questo e nel prossimo capitolo il problema complesso della gestione delle eccezioni in un programma wxPython, che come vedremo è in qualche modo interconnesso con le osservazioni sul *logging* che abbiamo fatto nei due capitoli precedenti. Potreste pensare che le eccezioni in wxPython funzionano come in Python, ma come vedremo le cose non stanno proprio così. Un programma wxPython è sempre il risultato dell'interazione di codice Python e codice C++ sottostante: e anche le coppie meglio assortite talvolta non vanno d'accordo.

Nel caso specifico della gestione delle eccezioni, ci sono tre aree problematiche:

1) le eccezioni Python non sono completamente libere di propagarsi in un programma wxPython: ne derivano alcune trappole insidiose che bisogna saper evitare;
2) non esiste un meccanismo di traduzione tra eccezioni C++ ed eccezioni Python...
3) ...ma servirebbe comunque solo fino a un certo punto, perché wxWidgets non usa le eccezioni C++ per segnalare condizioni di errore. wxWidgets fa uso di varie formule di *assert* e/o di allarmi emessi con il suo sistema di log interno. Esiste un meccanismo di traduzione degli *assert* C++ in eccezioni Python, e come abbiamo visto nel capitolo precedente si possono escogitare soluzioni per gestire meglio le scritture di log. Ma in entrambi i casi ci sono dei limiti.

Affrontiamo in questo capitolo il problema delle eccezioni Python. Il prossimo capitolo sarà dedicato all'analisi delle condizioni di errore che possono originarsi dal codice C++ di wxWidgets.

49.1 Il problema delle eccezioni Python non catturate.

Non è difficile accorgersi che in wxPython le eccezioni si comportano in modo anomalo. Un'eccezione non intercettata, in un normale programma Python, si propaga fino in cima allo *stack* senza trovare nessun `except` disposto a gestirla, finché il gestore di default non termina il programma inviando il *traceback* dell'eccezione allo *standard error* (e quindi, tipicamente, alla shell che ha invocato lo script).

In wxPython d'altra parte, un'eccezione non controllata *non termina il programma*:

```
class MainFrame(wx.Frame):
    def __init__(self, *args, **kwargs):
        wx.Frame.__init__(self, *args, **kwargs)
        b = wx.Button(self, -1, 'clic')
        b.Bind(wx.EVT_BUTTON, lambda evt: 1/0) # ops!
```

In questo esempio, ogni volta che clicchiamo sul pulsante si innesca una `ZeroDivisionError` non gestita. Il *traceback* relativo compare in effetti nello *standard error* (nella shell, quindi) ma l'applicazione wxPython resta perfettamente funzionante.

Per capire questo bizzarro comportamento occorre tenere presente che, durante il ciclo di vita di un'applicazione wxPython, il controllo dell'esecuzione passa di continuo da «strati» di codice Python a «strati» di codice wxWidgets (C++). Per esempio, quando l'utente fa clic su un pulsante innesca come sappiamo il complesso meccanismo dell'emissione di un evento e della ricerca di un gestore: questa fase è controllata da wxWidgets (C++). Quando un *handler* per l'evento è stato trovato, viene eseguito il *callback* relativo: questo in genere è codice che abbiamo scritto noi (Python). Quando il callback è stato eseguito, il controllo ritorna a `wx.App.MainLoop`: questo è di nuovo codice C++. E così via.

Ora, ecco il problema: una eccezione Python non può propagarsi oltre lo «strato» Python in cui viene emessa. Non è possibile propagare un'eccezione Python attraverso il codice C++. Questo è un problema di cui gli sviluppatori wxPython sono ben consapevoli: in teoria dovrebbe essere possibile tradurre al volo una eccezione Python nella sua controparte C++ e viceversa, in modo da permettere la propagazione libera tra i vari strati. In pratica però non è affatto banale implementare questo meccanismo: sono stati fatti dei tentativi, ma non si è mai approdati a nulla di definitivo.

E quindi? Che cosa succede quando l'eccezione Python si propaga fino al «confine» dello strato in cui è stata generata, senza trovare nessun blocco `try/except` disposto a prendersene cura? Succede una cosa brutta ma inevitabile: il codice C++ immediatamente successivo rileva che c'è una condizione di errore, chiama `PyErr_Print` per scriverlo nello *standard error*, e resetta l'errore. Quindi l'eccezione termina lì e non ha modo di propagarsi fino a raggiungere il punto in cui l'interprete Python farebbe arrestare il programma. Noi possiamo vedere ugualmente il *traceback* nella shell o dovunque lo abbiamo re-indirizzato, *come abbiamo visto* (pagina 278): ma questo solo perché è stato scritto lì da `PyErr_Print`, una delle API C di Python (https://docs.python.org/3/c-api/exceptions.html#c.PyErr_Print).

Infine, va detto che esiste un caso interessante in cui questo comportamento non si applica. Se l'eccezione Python non controllata avviene prima ancora di aver chiamato `wx.App.MainLoop`, allora effettivamente il programma si interromperà «come al solito»: questo perché, prima di entrare nel *main loop* di wxPython, è ancora Python ad avere il controllo dell'applicazione. Di solito ci sono due posti in cui un'eccezione Python può verificarsi prima di entrare nel main loop: in `wx.App.OnInit` e nello `__init__` della finestra principale dell'applicazione.

49.2 `try/except` in wxPython non funziona come ci aspettiamo.

Occorre essere consapevoli che il diverso comportamento delle eccezioni in wxPython può condurre a situazioni insolite per un programmatore Python. In Python «puro» possiamo intercettare un'eccezione anche molto lontano dal punto in cui si è generata:

```
>>> def disastro():
...     return 1/0  # ops!
...
>>> def produci_un_disastro():
...     return disastro()
...
>>> def prepara_un_disastro():
...     return produci_un_disastro()
...
>>> def salva_la_giornata():
...     try:
```

(continua...)

(...segue)

```
...         return prepara_un_disastro()
...     except ZeroDivisionError:
...         return "salvo per miracolo!"
...
>>> salva_la_giornata()
'salvo per miracolo!'
```

Questa non è una buona pratica di programmazione: ma si può comunque fare. In wxPython, invece, intercettare un'eccezione lontano dal punto di origine potrebbe non riuscire come ci aspettiamo:

```
class MainFrame(wx.Frame):
    def __init__(self, *args, **kwargs):
        wx.Frame.__init__(self, *args, **kwargs)
        p = wx.Panel(self)
        b = wx.Button(p, -1, 'clic', pos=(10, 10))
        b.Bind(wx.EVT_BUTTON, self.on_clic)
        self.Bind(wx.EVT_SIZE, self.on_size)                    ❶

    def on_clic(self, event):
        print('Nel callback del pulsante...')
        try:
            self.SendSizeEvent()                                ❸
        except ZeroDivisionError:
            print('presa al volo!')

    def on_size(self, event):
        print('Nel callback di EVT_SIZE...')
        event.Skip()
        1/0 # ops!                                              ❷
```

Qui abbiamo collegato (❶) `wx.EVT_SIZE` a un callback che produce una eccezione Python non gestita (❷). Possiamo aspettarci, tutte le volte che ridimensioniamo la finestra, di veder comparire il *traceback* nella shell. La finestra però si ridimensiona correttamente: a wxWidgets non interessa il *nostro* codice problematico: fintanto che chiamiamo `wx.Event.Skip`, l'handler di default dell'evento continuerà a fare il suo mestiere. Per la precisione, un primo `wx.EVT_SIZE` viene emesso automaticamente già al momento di creare la finestra: quindi dovremmo vedere un *traceback* nella shell proprio come prima cosa.

Quando però facciamo clic sul pulsante, ci aspettiamo una cosa diversa. Nel callback del pulsante noi generiamo manualmente un `wx.EVT_SIZE` (❸): quindi il callback difettoso verrà di nuovo eseguito, eccezione compresa. Questa volta però ci siamo premuniti, e abbiamo incluso la chiamata che genera il `wx.EVT_SIZE` in un blocco `try/except`. Dunque, ciò che vedremo questa volta sarà il messaggio «presa al volo!» nella shell, vero?

Purtroppo no. Il problema è che, tra lo strato di codice Python che innesca l'eccezione e lo strato di codice Python pronto a intercettarla, c'è di mezzo un consistente strato di codice C++ che si occupa del *dispatch* del `wx.EVT_SIZE`. E attraverso questo strato la nostra eccezione non passa. Il risultato è che, quando il `wx.EVT_SIZE` si genera in seguito al clic sul pulsante, il ramo `except` che abbiamo predisposto non sarà mai raggiunto dall'eccezione, a e noi non resterà che vedere comunque il *traceback* nella shell.

Non esiste una soluzione generale di questo problema. La cosa migliore è attenersi al noto principio di buon senso: le eccezioni dovrebbero essere intercettate il più vicino possibile al punto di emis-

sione. In Python è una buona pratica di programmazione, ma in wxPython è un principio guida da seguire con il massimo scrupolo.

49.3 Che cosa fare delle eccezioni non gestite.

Abbiamo visto che le eccezioni Python possono essere più difficili da intercettare in un programma wxPython, e abbiamo visto che le eccezioni non gestite non terminano prematuramente il programma. Questo però ci lascia con una domanda: come dovremmo comportarci con queste eccezioni non intercettate?

49.3.1 Il problema.

Prima di tutto, non dovrebbe esserci bisogno di dirlo: in un programma con interfaccia grafica, pensato per l'utente finale, le eccezioni non gestite sono *bachi*. Non dovrebbero esserci. Se capitano in fase di sviluppo, nessun problema: si tiene d'occhio lo *standard error* nella shell, si legge lo *stacktrace*, si corregge il baco.

Tuttavia, sappiamo che i bachi vengono fuori anche (o soprattutto!) quando ormai il programma è «in produzione». È a questo punto che wxPython ci fa rischiare di più. In un normale programma Python, un baco significa che l'eccezione non gestita ferma il programma. L'utente non sarà felice e forse l'uscita anomala lascerà i dati e qualche risorsa esterna in uno stato inconsistente. Almeno però il problema è immediatamente visibile e il programma si arresta senza che il danno abbia modo di propagarsi.

In un programma wxPython, d'altra parte, l'utente finale non vede lo *standard error* e non ha modo di accorgersi di nulla. Ecco uno scenario fin troppo comune (in pseudo-codice):

```
class Anagrafica(wx.Frame):
    def __init__(...):
        ...
        ok.Bind(wx.EVT_BUTTON, self.salva_dati)

    def salva_dati(self, event):
        dati = self._raccogli_dati()
        try:
            database.persisti(dati)
        except database.QualcosaNonVa:
            wx.MsgBox('Qualcosa non va')
            return
        wx.MsgBox('Dati salvati, tutto a posto')
        self.Close()
```

Quando l'utente fa clic sul pulsante «Salva», noi invochiamo una routine per salvare i dati nel database (`database.persisti` potrebbe far riferimento a un *ORM*, o comunque a un *layer* separato in una logica *MCV*). La nostra routine innesca un'eccezione *custom* in caso di problemi, e noi correttamente la intercettiamo nel callback. Dunque, se qualcosa non va, l'utente vede un messaggio allarmante. Se invece tutto va bene, l'utente viene rassicurato e la finestra si chiude. Questo pattern in sé non ha niente di sbagliato. Ma se c'è un baco nel *layer* di gestione del database, `database.persisti` innesca un'eccezione che non abbiamo previsto: siccome non la intercettiamo, tutto sembra andare per il verso giusto (si ricordi, wxPython non ferma il programma). Ma in realtà i dati non sono stati salvati. Prima che l'utente abbia modo di accorgersi del problema, l'errore potrebbe ripetersi più volte; i dati sbagliati saranno usati per ulteriori elaborazioni; e il baco originario potrebbe essere molto difficile da individuare.

Ora, in Python un approccio come questo viene giustamente considerato una cattiva pratica:

```
try:
    main()
except:  # un "bare except" per ogni possibile imprevisto
    # ...
```

Tuttavia possiamo chiederci se in wxPython non sia l'unico modo per risolvere il problema delle «eccezioni invisibili». Ci piacerebbe poter scrivere qualcosa come:

```
if __name__ == '__main__':
    # questo non funziona davvero!
    try:
        app = wx.App(False)
        MainFrame(None).Show()
        app.MainLoop()
    except:
        wx.MessageBox('Qualcosa non va!!')
        wx.Exit()
```

Purtroppo, come si può facilmente intuire, questo non funziona. A partire da quando invochiamo `wx.App.MainLoop` ci sono troppi strati di codice C++ perché una eccezione Python imprevista possa finire nella rete del *bare except* che abbiamo messo al livello più alto.

49.3.2 Una soluzione accettabile.

Una soluzione accettabile è invece usare `sys.excepthook` dalla libreria standard di Python (https://docs.python.org/3/library/sys.html#sys.excepthook):

```
def my_excepthook (extype, value, tback):
    wx.MessageBox('Questo non va proprio bene!', 'errore')
    # non dimentichiamo di loggare... qualcosa come:
    # logger.error('disastro fatale', exc_info=(extype, value, tback))
    wx.Exit()

class MainFrame(wx.Frame):
    def __init__(self, *args, **kwargs):
        wx.Frame.__init__(self, *args, **kwargs)
        p = wx.Panel(self)
        b = wx.Button(p, -1, 'clic', pos=(10, 10))
        b.Bind(wx.EVT_BUTTON, lambda evt: 1/0) # ops!

if __name__ == '__main__':
    import sys
    sys.excepthook = my_excepthook
    app = wx.App(False)
    MainFrame(None).Show()
    app.MainLoop()
```

Nel nostro *except hook* personalizzato possiamo inserire una logica arbitrariamente complessa. Per esempio, sarebbe perfettamente sicuro interagire con l'interfaccia grafica (aprire e chiudere finestre, postare eventi nella coda, etc.): la nostra eccezione Python non impedisce al framework C++ sottostante di continuare a funzionare, come sappiamo. Tuttavia, proprio perché stiamo affrontando un'eccezione imprevista (*un baco*) conviene limitarsi al minimo indispensabile: avvertire l'utente che il programma sta per chiudersi; fare il *rollback* di eventuali transazioni in corso; loggare il *traceback* dell'eccezione che altrimenti andrebbe perduto; infine, chiudere l'applicazione *nel modo che riteniamo migliore* (pagina 272).

Possiamo inserire il nostro *except hook* direttamente nel blocco `if __name__=="__main__"`, come nel nostro esempio. In alternativa, un buon momento è come sempre `wx.App.OnInit`.