Typesetting with Python

@brandon_rhodes
2019 June 16
PyLondinium

τέχνη
craft / art

TeX

markup language

plain text → document

```
\par
From the Black Speech, however, were derived
many of the words that were in the Third
Age widespread among the Orcs, such as
{\it gh\^ash} `fire', but after the first
overthrow of Sauron this language in its
ancient form was forgotten by all but
the Nazg\^ul.
```

From the Black Speech, however, were derived many of the words that were in the Third Age widespread among the Orcs, such as *ghâsh* ‘fire’, but after the first overthrow of Sauron this language in its ancient form was forgotten by all but the Nazgûl.

The book’s lessons
on how to type .tex files
were a small course in typography

```
Mr.~Baggins
Mrs.~Cotton
```

Hobbit-lore

1158–60

Stick to your plan—your whole plan

–π

Hobbit-lore
1158--60
Stick to your plan---your whole plan
$-\pi$

Math typesetting

The real reason for TeX:

When math journals stopped paying for professionals to set type by hand, math papers looked so ugly that Knuth could no longer publish

So he took an entire year off to invent TeX

```
$$
\sum_{k=0}^\infty
{(-1)^k \theta^{2k+1} \over (2k + 1)!}
= \sin \theta
$$
```

$$\sum_{k=0}^{\infty} \frac{(-1)^k \theta^{2k+1}}{(2k+1)!} = \sin\theta$$

Paragraphs

TeX represents the words of a paragraph
as fixed-width “boxes” separated by stretchy “glue”

A paragraph with n positions
at which the text could be split into lines
can be laid out in 2^n different ways

How could we ever find the optimium layout?

Dynamic programming!

TeX finds the optimal solution
for breaking each paragraph into lines

$O(n^2)$ worse case, usually $O(n)$
(n = number of possible breaks)

```
\par
From the Black Speech, however, were derived
many of the words that were in the Third
Age widespread among the Orcs, such as
{\it gh\^ash} `fire', but after the first
overthrow of Sauron this language in its
ancient form was forgotten by all but
the Nazg\^ul.
```

From the Black Speech, however, were derived many of the words that were in the Third Age widespread among the Orcs, such as *ghâsh* ‘fire’, but after the first overthrow of Sauron this language in its ancient form was forgotten by all but the Nazgûl.

The output of TeX was beautiful!
But it was difficult to control.

Once you set up the parameters,
layout proceeded largely outside
of your control

Backing up a tractor and trailers
is an open problem in AI

“Fuzzy Knowledge-Based Control for Backing Multi-Trailer Systems”

Andri Riid, Jaakko Ketola, Ennu Rüstern

Trailers are difficult to back up
because the input — the motion of the tractor —
has an increasingly distant relationship
to the motion of the nth trailer

Trying to control TeX
sometimes felt similar

Idea:

What if instead of typesetting
“systems” that we merely configure

there were a typesetting “library”
that left the programmer in control?

Recently,
I realized that typesetting
and printing a book from Python
was coming within reach!

Print-on-demand

PDF → custom hardcover

Real hardcover!

- Casebound
- Smyth sewn

Technology

MetaFont → TrueType, OpenType
Macro language → Markdown, RST
Paragraph layout → Andrew Kuchling’s texlib
DVI → PDF

But what would I print?

My grandfather’s essays

And I would write the typesetting myself

And I would write the typesetting myself
— in Python!

Hwæt!

Hwæt!

Re-implementing TeX

Hwæt!

What would I do differently?

I chose a specific first goal

Different width columns?
Not supported in TeX

As TeX breaks a paragraph into lines
it doesn’t even know what page
the paragraph will land on

paragraph → lines
is a separate step from
lines → pages

My idea: the paragraph should ask for more space as it needs it, so it learns about any width change when it crosses to a new column

Plan

1. Find a library for rendering PDF
2. Invent a new page layout engine

ReportLab

He was named for two Revolutionary War heroes

Was there an alternative?

ReportLab

He was named for two Revolutionary War heroes

Qt

He was named for two Revolutionary War heroes

Input: list of typesetting actions

```
# Input (Markdown, RST, etc) produces:
actions = [
    (title, 'Prologue'),
    (heading, '1. Concerning Hobbits'),
    (paragraph, 'Hobbits are an unobtrusive…'),
    (paragraph, 'For they are a little people…'),
    (heading, '2. Concerning Pipe-weed'),
    (paragraph, 'There is another astonishing…'),
]
```

What API should the layout engine use to call each action?

Let’s start by asking:
what information does
an action need?

```
Column = NamedTuple(…, 'width height')
paragraph(column, y, ...)
```

```
# Each time the paragraph needs another line:
leading = 2pt
height = 12pt
if y + leading + height > column.height:
    # ask for another column
```

Q: How do we ask for another column?

```
def paragraph(column, y, ...):
    column2 = column.next()

def paragraph(column, y, layout, ...):
    column2 = layout.next_column(column)

def paragraph(column, y, next_column, ...):
    column2 = next_column(column)
```

A: Pass a plain callable!

```
def paragraph(column, y, next_column, ...):
    column2 = next_column(column)
```

Why?

To avoid
premature
Object Orientation

“Premature optimization is the root of all evil”

“Premature optimization is the root of all evil”
— Donald Knuth

Premature Object Orientation:

attaching a verb to a noun
when you don’t need to yet

Symptom:

Passing an object
on which a function will
only ever call a single method

```
def paragraph(column, y, layout, ...):
    column2 = layout.next_column(column)
```

Premature Object Orientation
couples code that needs only a verb
to all the implementation details of the noun

```
def paragraph(column, y, next_column, ...):
    column2 = next_column(column)
```

```
# So now I had a rough plan for action inputs:

def paragraph(column, y, next_column, ...):
    column2 = next_column(column)

# What would an action return?
```

Can the paragraph
simply go ahead and draw
on the output device?

No

Problem:
Headings

A heading is supposed to sit atop
the content of which it is the head

and all but Hobbits would find them exceedingly dull. Hobbits delighted in such things, if they were accurate: they liked to have books filled with things that they already knew, set out fair and square with no contradictions.

2. Concerning Pipe-weed

There is another astonishing thing about Hob-

Q: What if there’s no room beneath the heading?

A: Typographic disaster

and all but Hobbits would find them exceedingly dull. Hobbits delighted in such things, if they were accurate: they liked to have books filled with things that they already knew, set out fair and square with no contradictions.

2. Concerning Pipe-weed

The heading needs to move itself to the next column

```
# Can the Heading simply check whether
# there is room for a line beneath it?

if y + 2 * (leading + height) > column.height:
    column = next_column(column)
    y = 0
```

But checking for a free line
won’t, alas, always work

Why?

Because a paragraphs might not choose to use the final line of a column!

“widows and orphans”

A single-line paragraph might deign
to remain at the bottom of the page

and all but Hobbits would find them exceedingly dull. Hobbits delighted in such things, if they were accurate: they liked to have books filled with things that they already knew, set out fair and square with no contradictions.

2. Concerning Pipe-weed

There is another astonishing thing.

But a multi-line paragraph will
refuse to leave its opening line alone —
will refuse to leave it an “orphan”

and all but Hobbits would find them exceedingly dull. Hobbits delighted in such things, if they were accurate: they liked to have books filled with things that they already knew, set out fair and square with no contradictions.

2. Concerning Pipe-weed

There is another astonishing thing about Hob-

and all but Hobbits would find them exceedingly dull. Hobbits delighted in such things, if they were accurate: they liked to have books filled with things that they already knew, set out fair and square with no contradictions.

2. Concerning Pipe-weed

How can the heading predict whether it will be stranded alone?

How can the heading predict
whether it will be stranded alone?

(a) Know everything about paragraphs

How can the heading predict
whether it will be stranded alone?

(a) Know everything about paragraphs
— or —
(b) Ask next action to lay itself out speculatively

But this is going to require
“undo” — the ability to back up

```
# Heading

def heading(...):
    add itself to document
    add the following paragraph to the document
    if there is content beneath heading:
        return
    undo the paragraph
    undo the heading
    start over on next page
```

Consequence #1

Layout needs to return
an intermediate data structure
that the caller can inspect

Consequence #2

The intermediate data
needs to be easy to discard

Iterators?
Generators?
Lists of lists?
Trees?

A:

A:
Linked list

```
Column = NamedTuple(… 'width height')
Line = NamedTuple(…, 'previous column y graphics')

c1 = Column(…)
line1 = Line(None, c1, 0, [])
line2 = Line(line1, c1, 14, [])
line3 = Line(line2, c1, 28, [])
```

$\boxed{\texttt{line}} \leftarrow \boxed{\texttt{line}}$

$$\begin{tikzcd}
 & & \boxed{\texttt{line}} \arrow[dl] & \boxed{\texttt{line}} \arrow[l] & \boxed{\texttt{line}} \arrow[l] & \text{A} \\
\boxed{\texttt{line}} & \boxed{\texttt{line}} \arrow[l] & & & & \\
 & & \boxed{\texttt{line}} \arrow[ul] & \boxed{\texttt{line}} \arrow[l] & \boxed{\texttt{line}} \arrow[l] & \text{B}
\end{tikzcd}$$

A linked list lets us extend the document with any number of speculative layouts, which Python automatically disposes of as we discard them

Actions now need a new argument:
the most recently laid out line

```
           ↓
paragraph(line, column, y, next_column, ...):
```

But look!

```
                  ↓     ↓
paragraph(line, column, y, next_column, ...)

# What does a line know?         ↓    ↓
Line = NamedTuple(…, 'previous column y graphics')
```

```
paragraph(line, next_column, ...)
```

Designing our return value
wound up eliminating two
of our input arguments

Always look for chances to simplify
as you proceed with a design

Also nice:
Symmetry!

```
# The Line becomes a common currency that is
# both our argument and our return value:

def paragraph(line, next_column, ...):
    ...
    return last_line_of_paragraph
```

But:
(a) How will the heading action
invoke the action that follows?

(b) How it will tell the engine
that the following action
is already laid out?

Special callable?
Exception?
Coroutine?

```
# Input (Markdown, RST, etc) produces:
actions = [
    (title, 'Prologue'),
    (heading, '1. Concerning Hobbits'),
    (paragraph, 'Hobbits are an unobtrusive…'),
    (paragraph, 'For they are a little people…'),
    (heading, '2. Concerning Pipe-weed'),
    (paragraph, 'There is another astonishing…'),
]
```

```
def heading(actions, a, line, next_column, …):
    …
    return a + 2, line_n
```

Incrementing and returning the `a` index lets an action invoke as many subsequent actions as it needs to

Stepping back, I looked askance at
the amount of repetition in my code

```
# opinionated
def heading(actions, a, line, next_column, …):
    … return a + 2, line2
def section(actions, a, line, next_column, …):
    … return a + 3, line2

# simple
def paragraph(actions, a, line, next_column, …):
    … return a + 1, line2
def center_text(actions, a, line, next_column, …):
    … return a + 1, line2
```

For “opinionated” actions
that care about what follows
it’s necessary to pass
`action` and `a`

But simple actions ignore them!

```
# opinionated
def heading(actions, a, line, next_column, …):
    … return a + 2, line2
def section(actions, a, line, next_column, …):
    … return a + 3, line2

# simple
def paragraph(actions, a, line, next_column, …):
    … return a + 1, line2
def center_text(actions, a, line, next_column, …):
    … return a + 1, line2
```

How can I eliminate `actions` and `a` from simple actions that don’t need them?

DRY

"Don't Repeat Yourself"

I suddenly
heard the call
of distant decades

1990s

Introspect each function to learn
if it takes `actions` and `a` or not!

```
def heading(actions, a, line, next_column, …):
    … return a + 2, line2
def section(actions, a, line, next_column, …):
    … return a + 3, line2

def paragraph(line, next_column, …):
    … return line2
def center_text(line, next_column, …):
    … return line2
```

Early 2000s

Special registry for functions
that don’t need `actions` and `a`

Late 2000s

A decorator for functions
that don’t need `actions` and `a`

```
def simple(function):
    def wrapper(actions, a, line, next_col, *args)
        line2 = function(line, next_col, *args)
        return a + 1, line2
    return wrapper
```

```
def heading(actions, a, line, next_column, …):
    … return a + 2, line2
def section(actions, a, line, next_column, …):
    … return a + 3, line2

@simple
def paragraph(line, next_column, …):
    … return line2
@simple
def center_text(line, next_column, …):
    … return line2
```

DRY

And what did I decide?

I decided
to repeat myself

```
# opinionated
def heading(actions, a, line, next_column, …):
    … return a + 2, line2
def section(actions, a, line, next_column, …):
    … return a + 3, line2

# simple
def paragraph(actions, a, line, next_column, …):
    … return a + 1, line2
def center_text(actions, a, line, next_column, …):
    … return a + 1, line2
```

Why?
Symmetry

```
# opinionated
def heading(actions, a, line, next_column, …):
    … return a + 2, line2
def section(actions, a, line, next_column, …):
    … return a + 3, line2

# simple
def paragraph(actions, a, line, next_column, …):
    … return a + 1, line2
def center_text(actions, a, line, next_column, …):
    … return a + 1, line2
```

When I return to code
months and years later
I re-learn by re-reading

Given a stack of functions
that do exactly the same thing,
if ½ of them use one convention
and ½ use another —

— then I now have twice
the number of conventions to re-learn,
and only half the number of examples
of each to learn from!

I chose verbose symmetry
over asymmetric brevity

As a reader,
I need routines
that behave the same
to look the same

```
# opinionated
def heading(actions, a, line, next_column, …):
    … return a + 2, line2
def section(actions, a, line, next_column, …):
    … return a + 3, line2

# simple
def paragraph(actions, a, line, next_column, …):
    … return a + 1, line2
def center_text(actions, a, line, next_column, …):
    … return a + 1, line2
```

We’re ready for a final design step!

widows
and
orphans

How does that look in code?

```
def paragraph(…):
    lay out paragraph
    if it stranded an orphan at the page bottom:
        try again
    if it stranded a widow at the page top:
        try again
```

Inside of its widow-orphan logic,
paragraph() had a hidden inner routine
that did the actual paragraph layout

What if you just wanted
to call the simple part?

```
def paragraph(…, no_widows=True, no_orphans=True):
    …
```

But Boolean switches are often
a hint that we have coupled what could
actually be two different routines

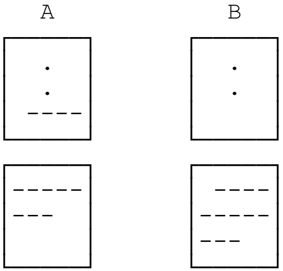
Composition » Coupling

```
actions = [
    (heading, '1. Concerning Hobbits'),
    (paragraph, 'Hobbits are an unobtrusive…'),
]
```

```
actions = [
    (avoid_widows_and_orphans,),
    (paragraph, 'Hobbits are an unobtrusive…'),
]
```

Avoiding an orphan?

Easy!

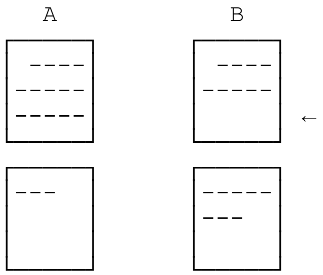

A
B

```
# Before calling the paragraph, simply:

column = next_column(column)
y = 0
```

Avoiding a widow?

Nearly impossible


A
B
←

How would we ever convince a paragraph
to move to the next column early?

```
# Each time the paragraph needs another line:

leading = …
height = …
if y + leading + height > column.height:
    # ask for another column

# How would we influence this choice?
```

Lie about the value of `y`?

Provide a fake column height?

We are looking desperately
for parameters to tweak because
we’re standing outside of the code
that makes the decision

Outside

Is that really where we want to be during a crucial decision?

No

When code is making a crucial decision —

You want to be
in the room where it happens

Right now, the paragraph only consults us when it needs a whole new column

`next_column()`

```
def next_line(line, leading, height):
    column = line.column
    y = y + leading + height
    if y > line.column.height:
         column = next_column(line.column)
         y = height
    return Line(line, column, y, [])
```

```
# What if the paragraph calls back not only
# when it *thinks* it needs a next_column()
# but every time it needs a next_line()?

def paragraph(…, next_line, …):
    …
```

Then, the widow-orphan logic
can subvert the paragraph's normal decision
simply by passing a custom next_line()!

```
def avoid_widows_and_orphans(…, next_line, …):

    def fancy_next_line(…):
        # A wrapper around next_line() that jumps
        # to the next column early!

    paragraph(..., fancy_next_line, ...)
```

Success!

Did you catch why
this was a success?

The fancy_next_line() wrapper is so simple
because we avoided premature Object Orientation!

```
# What if instead of just passing next_line()
# we were passing a whole Layout object?

def paragraph(..., line, layout, ...):
    line2 = layout.next_line(line)
```

How would you make an object's
next_line() method return a different value?

Monkey patching?
An Adapter class?
Gang of Four Decorator?

In Object Orientation, customizing a verb
can require trundling out an entire design pattern

But if you pass callables —
if you treat your verbs as first class citizens —
a simple inline wrapper can put you
in the room where it happens

Lessons

Start verbose, simplify later
Value symmetry over special cases
Avoid premature Object Orientation
Let verbs be first-class citizens

I plan on releasing my “typesetting” Python library later this summer — but you can already watch my progress on GitHub:

github.com/brandon-rhodes/python-bookbinding

Sure Print and Design
Toronto, Canada

Will print runs of only 2 books!

328 hardcover pages

Thank you very much!

@brandon_rhodes