Cubicle

Nov 16, 2020

Contents

Generate beautiful spreadsheets automatically, today.

Project Cubicle is a high-level declarative domain-specific language for fully-functioning, professional-looking, business-oriented numerical and graphical reporting.

CHAPTER 1

Introduction

This document lays out the big-picture ideas so you can get your bearings and understand how to exploit *cubicle*.

Bear in mind that some (decreasing) fraction of this document is aspirational: it describes where I'd like the project to get to, even though not everything is implemented yet.

1.1 Concept of Operations

A domain-specific language is defined to succinctly describe the structure, formatting, formulas, and boilerplate for business-oriented reports. An application can then produce any such report conveniently by supplying one or more streams of detail data and an environment of relevant business-domain knowledge such as collation sequences, entity attributes, and so forth.

The expectation is that the underlying business domain data model will be considerably more stable than the shifting winds of taste and temperament that dictate how reports should look and act. As requirements evolve, this kind of a system should help the programmer keep up. In particular, when a new report is conceptually similar to an old one, little more than a new skin is required: the domain model almost never changes, and existing data access methods are often sufficient for the new task.

A simple standard interface is defined for passing data into a report, agnostic to the source of that data. The package generates output into spreadsheet documents using the excellent *xlsxwriter* package by John McNamara. Once a report is generated, it can also be queried for the row- and column-extent of specific (sets of) elements for integration with additional requirements not specifically covered within this package.

Version one of the package focuses on attractive and professional presentation for tabular and hierarchical reports in one or two dimensions, with limited support for calculated functions. Later versions may add support for more kinds of smart functions, charting integration, different output formats such as HTML tables, ragged axes, or whatever else seems valuable.

1.2 Data Streams

At a Python level, *cubicle* mostly deals in streams of *point-magnitude* pairs. But these are no ordinary pairs: each *point* is in fact a dictionary!

In concept, the *point* names a specific location in a notional hyperspace; but each hyperspatial dimension has a name (rather than a positional index) so a *point* is a dictionary from dimension-names to *ordinal* values. The *magnitude* contributes its value to the hyperspatial location named by the *point*.

An *ordinal* naturally ought to be drawn from what's appropriate given the type of its dimension. For the moment, that restriction is up to you to follow, because violating it will just break things later. For examples:

- a "Time" dimension might have `datetime.date` objects for ordinals.
- a "Grocery" dimension might have SKU numbers as ordinals.
- a "Salesperson" dimension might have employee IDs as ordinals.

In some cases, you'll deal in non-numeric *magnitude* data. This is particularly relevant with list-of-entity style reports.

Where you get your data streams from is up to you. However, relational database queries are likely to be a good start.

Along with a stream of *point-magnitude* pairs you can generally specify a context, which is another *point* along a different set of dimensions. That can be useful particularly for routing the results of different queries into different portions of a report grid.

1.3 The Report "Canvas"

A canvas has a horizontal and vertical layout structure, as well as a private data store and a runtime-environment, which supplies certain supporting functions. As a 3-D alternative, you might possibly want a family of canvases, sharing structure but using different private data for each.

An axis respects a particular layout elements (from the symbol table) and manages tree of hash keys at run-time. It exposes operations to find the hash node corresponding to a particular data point, and also to resolve node selection expressions (explained later).

Actually, the bulk of the work to resolve those queries must be delegated to the layout elements, because the layout elements refer to *Reader* objects and potentially other custom bits.

Reader objects come in two forms: * *SimpleReader* plucks a domain value directly from some input *point*. * *ComputedReader* passes a point to a Python function registered with the *Grid*'s environment object. This is particularly useful for automatic implied categorization.

CHAPTER 2

Language Overview

This document lays out the syntax and semantics of the `cubicle` report specification language.

2.1 Overview

The general idea is you define your report "skins" in a *cubicle* module as described herein, compile that module to a static data structure (which you may pickle) and then later, construct reports based on those skins with data you supply at runtime. Details of those other operations are in the chapter on integration.

2.1.1 Top-Level Definitions

At the outermost (top-level) syntactic level, a cubicle module defines:

- **layout structures** These provide the general structure, format, and boilerplate for a report in one direction (either horizontal or vertical) or may be used as sub-components in larger layout structures.
- **styles** These provide for naming and re-using particular groups of color, font, border, formatting, etc. etc. etc. Styles may be used anywhere formatting directives are appropriate.
- **canvas definitions** These describe entire report structures in 2-D. They:
  - refer to horizontal and vertical layout structures,
  - provide global "background" formatting, and
  - provide "patch" boilerplate: things specified in both the horizontal and vertical – or at any rate things specified to override the default interaction between rows and columns.

Every definition begins with a name (identifier), then a keyword describing what sort of thing is being defined.

2.1.2 Contributing Components

Within the top-level definitions, certain other things may be defined:

- **Subordinate Layout** Layout structure syntax allows arbitrary in-place nesting, so in general things that conceptually go together are lexically together in the module file.
- **Named Routes** These abstract over cosmetic details of layout structure. They're defined within layout structure, and used within both selectors and the API for named reference deep into nested layout structures. (Or: They *will* be, when finished.)
- **Selectors** Every place that needs to address a portion of other layout uses the same syntax and supports pretty much the same ideas.
- **Template Strings** Anything inside double quotes supports various automagical substitutions.
- **Formula Strings** Formulas in spreadsheets refer to other cells. *Cubicle* abstracts out the specific row and column numbers, so formulas can contain selectors to pick out the cells you mean, symbolically.
- **Patch Instructions** These take care of all the special exceptional cases in your report layouts which cannot be expressed simply as the cross-product of horizontal and vertical layout structures.

2.2 Tokens or Lexemes

The cubicle module language is composed of:

- **Keywords**

  All start with a colon and delineate grammatical structures. These are:
  `:axis :canvas :frame :gap :head :leaf :menu :merge :path :style :tree :use`
  Keywords are not case-sensitive.
- **Identifiers** Following the usual programming-language convention, these start with a letter and may contain digits and underscores. Certain identifiers are special in certain contexts.
- **Sigils** Punctuation marks prefixing an identifier to inflect it with special meaning:
  - `%foo` is a style reference.
  - `@foo` is a computed reference. (You supply a definition at runtime).
  - `+foo` and `-foo` turn on or off boolean formatting elements like bold or underline.
  - `~foo` is a reference to a named route in the layout.
- **Integer and Real numbers** These follow the ordinary conventions for representation. In addition, you can supply a hexadecimal integer by prefixing it with the `$` sign, as in `$DEADBEEF`.
- **Colors** In addition to the sixteen predefined color names, a hash mark followed by six hexadecimal digits, like `#feeded` is taken as a color. This rule takes precedence over the end-of-line comment rule.
- **Comments** a hash mark (`#`) which is NOT immediately followed by six hexadecimal digits is taken as the start of a comment, which extends to the end of the same line. Comments are ignored, like whitespace.
- **Simple strings** surrounded by single quotes `'like this'`, and which do not implement substitution.
- **Template strings** surrounded by double quotes `"like this"` and which interpolate substitution parameters found within `[square]` brackets. There's a modicum of structure available within such parameters for addressing different bits and bobs of information.
- **Formula strings** prefixed by the (`@`) sign and otherwise surrounded by single quotes like this example: `@'sum([across=_,winner=*,victory=*])'` Square brackets delimit reference-replacement parameters.

- **Whitespace** Newlines are significant to the general syntax. Horizontal whitespace is taken literally within all kinds of strings. In other respects, the amount and type of whitespace is ignored except as a a convenient separator between tokens which might otherwise be confused. In particular, indentation is not significant, but it's good for anyone reading your code.
- **Various punctuation and nesting concepts** Commas, semicolons, curly braces, brackets, and parentheses all have their places.

2.3 Ideas for the Future

It's entirely possible new features could be added. If you have a good suggestion, please send it in. You should be able to contact me through GitHub.

CHAPTER 3

Styles

All layout elements, canvases, and patch definitions can be styled with formatting attributes, which are basically defined by what the *xlsxwriter* module supports.

The language allows you to define and refer back to named collections of formatting attributes.

3.1 Format Attributes

You can use sigils like `+bold` and `-text_wrap` turn on or off boolean attributes. Other attributes need a specific value. If that value is a number or *looks like an identifier,* you may supply it without quotes, as in `align=center`. If the attribute is a complex string, surround it with single-quotes, as in this example: `num_format='0.0%;[red]-0.0%'` Finally, if the attribute is a color, you can use either a predefined name, like `font_color=green` or a hexadecimal color code, like `bg_color=#ffcccc`. (At some point support for decimal RGB colors may be added.)

List of Pre-Defined Colors:

black blue brown cyan gray green lime magenta
navy orange pink purple red silver white yellow

The exact list of supported attributes is defined in the file `cubicle/xl_schema.py`, which please see.

Note on special cases: Setting either of the properties *border* or *border_color* stands in for setting the corresponding attributes on all four of *top, bottom, left*, and *right*.

3.2 Defining a named style

To define a style called “example”, include a line like:

```
example :style +bold +underline align=center
```

The pattern is:

1. name of the style

2. keyword `:style`
3. one or more formatting attributes as described in the previous section.
4. newline

Please note: styles can only be defined in the outermost scope of a module. Attempting it nested inside other structures will yield a syntax error upon compiling the module.

3.3 Referring back to a style

Assuming you've defined a style called "example", then later on in the module you can refer back to it with the `%example` sigil wherever formatting attributes are appropriate, *including in subsequent style definitions*.

CHAPTER 4

Layout Structures

Layout structures declare the general idea of how a report should be laid out. Any given report will have one horizontal and one vertical layout structure. The structures come in several varieties which can be mixed and matched to form whatever layout you need.

A *cubicle* module can contain arbitrarily many layout structure definitions. The nesting structure of layout elements is normally given literally (in-place) but may instead refer back to previously-defined structures whenever that suits you.

4.1 The "marginalia" concept

Any given bit of layout is associated with various bits of information about boilerplate and formatting. Collectively, these data are called "marginalia". Think of them as notes scrawled in the margins. All of these notes are optional, but in the prescribed order of their appearance, they are:

1. **Header Text(s):** Either a string, a template, or a collection of these inside of parenthesis. These will be used according to formula hints on the perpendicular – explained later.

2. **Formula Hint** This may be any of:

   - A formula-string `@'like this'` which gives a formula to appear in the data cells along this row or column. This may optionally be followed by a priority specification, which a `@` followed by integer to breaks ties between row and column formulas.

     > If row and column both specify a formula, the higher priority number wins. The default priority is zero. If row and column formulas are tied for priority, the column formula wins. You can also apply formulas to patches declared inside a canvas definition, and these take precedence over everything else.

   - `:gap` prevents most text from being written to this row or column, even by formula hints from the perpendicular. (However, header text prevails if supplied for this node.)

   - `:head 1` populates the row or column with the first (if any) header text drawn from the perpendicular marginalia. If those marginalia have multiple header texts, replace the `1` with the appropriate index.

     Headers called forth in this manner take precedence over formula strings.

3. **Formatting attributes and/or style references** These are as described in the section on styles, above.

   Where row and column formats set different values on the same attribute, the column formatting prevails. You can also apply formats to patches declared inside a canvas definition, and these again take precedence.

Layout definitions have somewhat of a tree structure to them. Marginalia established at a parent node automatically applies to all child nodes unless a child expressly changes something.

4.2 Leaf Nodes

Leaf nodes are the smallest (atomic) unit of layout. They represent either a single row or column without any dependence on data. They can carry the full complement of marginalia.

4.2.1 Inside larger structures

Inside larger structures, the presence of a leaf node is implied by not overtly declaring the use of some other kind of sub-structure. You would instead simply supply the appropriate marginalia (as described above) wherever the syntax calls for a subordinate structure definition, and *cubicle* will do the right thing.

4.2.2 Stand-alone (named) leaves

There are a couple reasons you might wish to define a leaf node as a top-level named structure. One idea is when you want to emit a one-dimensional report – that is, a report with either a single row or a single column. No matter: your reasons are your own. If you want to do it, *cubicle* makes it possible.

To name a leaf-node as a module-level structure, give:

1. name of the structure,
2. keyword `:leaf`
3. whatever marginalia applies, as described above
4. newline

4.3 Composite Structures:

The composite structures are `:frame`, `:tree`, and `:menu`. They all split layout into parts according to a slightly different philosophy.

4.3.1 The Characteristic Axis

Composite structures split layout into parts. How shall the system determine which part we're addressing when?

When you're feeding data to a report, you supply <*point, magnitude*> pairs. The *point* is a dictionary (or mapping).

A composite structure's *reader* tells how to get the ordinal from whatever *point* is passed into the system. A reader also has a characteristic axis name.

A normal reader just uses the characteristic axis name as a key in the *point* dictionary: the corresponding value provides the ordinal used by the layout. A computed reader gets the ordinal values in a more roundabout way, explained in detail in the chapter on integration with Python.

It’s possible to supply a *reader* in three ways. The reader is:

- **By default,**

  normal, with the name of the corresponding layout structure.

  Example: `foo :tree` then `foo` is the reader for the tree called `foo`.

  But see the note on tree subordinates, later on.

- **`:axis` <name>**

  the reader is exactly the given name.

  Example: `foo :tree :axis bar` then `bar` is the reader for the tree called `foo`.

- **`:axis` <computed-sigil>**

  computed, with name equal to the bare name of the sigil.

  Example: `foo :tree :axis @bar` then the characteristic axis is `bar` but the system expects the runtime integration to supply a special method for computing `bar` ordinals from whatever *point* dictionaries get passed along in data streams.

4.3.2 Frames

> *name* `:frame` *[reader] marginalia* `[`
>
> > *field*
> >
> > ...
> >
> > *field*
>
> `]`

OR:

> *name* `:frame` *[reader] marginalia* `[` *field* `;` ... `;` *field* `]`

A frame splits layout into a fixed set of parts in exactly the order given. To route data among the parts, most normally you would supply the frame’s *name* as a key in the *point* of a <*point,magnitude*> pair, with corresponding value drawn from among the member field names.

Each *field* consists of a *name*, optionally a *path tag*, and a subordinate structure associated to that field. As a special exception, at most one *field* may have the name of _ which means to use this field by default whenever a point does not have an ordinal for this frame’s key. However, a composite subordinate to _ must have an `:axis` given explicitly, for it has no default name to fall back on.

> Path tags are a new feature under development at the moment. There is a separate section of this chapter devoted to them.

4.3.3 Trees

name `:tree` *[reader] marginalia substructure*

A tree splits layout into arbitrarily many parts, each with homogeneous substructure, according to the ordinals actually observed in the data stream on the characteristic axis.

Trees do not have fields, so originally they passed their own field-name as default axis-key to their substructure. This changed in version 0.8.5 to prepend `per_` to the tree’s own axis. For example, given something like

```
foo :tree :frame [ a; b ]
```

the *tree* has axis `foo`, but the *frame* has axis `per_foo`. You can of course override all this by sprinkling `:axis` phrases into appropriate places.

4.3.4 Menus

Menus provide adaptive ragged structure.

Menus have a syntax similar to that of frames, except using `:menu` in place of `:frame`. The semantics are different, though: First, a menu's fields only appear in the output report if their corresponding ordinals got mentioned in a data stream. Second, a menu may not have a field called _, because that would make no sense.

4.3.5 Defining Named Zones

Concept: Named zones attach a name to a specific section of a layout structure, for later reference elsewhere as a shorter, more shelf-stable alternative to the equivalent list of axis criteria.

This should:

1. Make other parts of a module definition less sensitive to cosmetic changes in layout.
2. Simplify references in formula strings and patch selectors.
3. Expose data routing information back to the run-time in a symbolic manner, making also the API less sensitive to irrelevant details of layout.

Defining Syntax: Immediately after a field's name in a *frame* or *menu* definition, the keyword `:zone` followed by an identifying name for the route's symbol.

Zone definitions must be unique within each distinct top-level layout definition.

4.4 Referring to defined structures

In place of a subordinate structure, `:use` <*name*> will evaluate to a copy of the named structure declarations.

For example:

```
foo :frame [p; d; q]
bar :frame [
        x :use foo
        y :use foo
        z +bold :use foo
]
```

This will cause all three elements of the `bar` frame to have substructure corresponding to the `foo` frame. In addition, the `+bold` format attribute applies to the `z` field.

4.4.1 Named Zones in Referred/Factored Structures

This is easiest to explain by example. Suppose we have a couple of layout structures something like:

```
inner :frame [
        quantity
        rate
        total :zone frobozz @'[inner=quantity]*[inner=rate]'
```

(continues on next page)

(continued from previous page)

```
]

outer :frame [
        product
        original :use inner
        current :use inner
        delta :use inner @'[change=current]-[change=original]'
]
```

In this example, the `inner :frame` contains a definition of `:zone frobozz`. Subsequently, the `outer :frame` makes three distinct references to `:use inner`.

Within the scope of the `outer :frame` (and anything using it) `~frobozz` is effectively defined as `inner=total` – exactly the same definition as applies within the `inner :frame` scope. It's (currently) an error to also declare `:zone frobozz` within the text of the `outer :frame`

Is this the be-all end-all answer? No, probably not. But it does enough of the job for now. If you have a good use-case why the semantics should be adjusted, please share.

CHAPTER 5

Canvas Definitions

The complete definition for the “skin” of a report is given by a canvas definition. This is what everything else builds up to.

5.1 Main Grammar Pattern

```
name :canvas across down formatting [
    patch
    ...
    patch
]
```

The given *name* is how you look up the canvas definition in the compiled *cubicle* module. (See the integration chapter for more.)

The identifiers *across* and *down* refer to (elsewhere-defined) layout structures.

The *formatting* is zero or more background-level format attributes. These apply to every cell in the report, but at the lowest conceivable priority.

5.2 Patch Instructions

Patch instructions are how you tweak the skin in ways you can’t express as the intersection of marginalia.

The general idea is that patches take effect as if painted in order from first to last. (That’s not the actual algorithm, but it could be, and the only distinction would be performance.)

5.2.1 Simple Patches

A simple patch instruction consists of:

<selector> { *<content>* *<formatting>* }

1. **Selector** a comma-separated list of selection criteria, explained below.
2. **Optional Content** If present, gives content to fill into the selected cells. This may be any of:
   - absent, which leaves cell content as-is.
   - a string of any sort (plain, template, or formula) which replaces the content of cells in the usual manner.
   - the `:gap` keyword, which expressly blanks out cells.
3. **Formatting** Any formatting attributes given here apply to all selected cells. These follow the same syntax as described in the section on styles.

5.2.2 Merge Patches

It's common to want to merge a block of cells together. The grammar for this is the `:merge` keyword in front of a simple patch:

`:merge` *<selector>* { *<content>* *<formatting>* }

The component parts work exactly as they do for a simple patch, but the selected cell blocks get merged in the report.

Hint: Say `something=*` in the selector to merge one block for each *something*.

5.2.3 Nesting Patches

New in version 0.8.8

Suppose several successive patch instructions have several criteria in common: It would be nicer to give the common subset of criteria, then nested within square brackets, a subordinate list of (now shorter) patch instructions.

```
<selector> [
    <patch>
    ...
    <patch>
]
```

To that end, this grammar pattern has been added and made to work recursively: you can nest selector contexts as deep as you like, although at some point you run out of things to specify.

CHAPTER 6

Selectors

A *selector* is a symbolic reference to some specific portion of layout. Selectors are used in a couple different ways. A selector:

Within a formula string, inside square brackets: Becomes a cell reference embedded in the resulting formula that gets written into the workbook when the canvas gets plotted.

In the head of a patch instruction: Tells which portion of the layout canvas to apply the templates, formulas, and formatting in the body of the patch instruction.

6.1 Selector Syntax and Semantics

Syntactically, a selector is written down as a comma-separated list of *criteria*. Semantically, it represents all cells (or in the case of merge-instructions, all cell-blocks) with layout-addresses that satisfy the conjunction (logical-AND) of the given criteria.

6.1.1 Ordinary Criteria

Each criterion is generally written as:

> <*axis*> = <*predicate*>

The *axis* is the name of a characteristic axis for some composite layout structure. (Even if the axis is computed, leave off the `@` inside a selector.)

6.1.2 Named-Zone Criteria

You may also refer to named-zones within selectors:

```
~like_this
```

The sigil ~ denotes a zone/route defined within layout structure. For example, `~hours` would refer to a route called “hours”, and stand in for all appropriate criteria to select that portion of layout.

Note Regarding Zone Intersections:

If both the horizontal and vertical layout structures associated with a `:canvas` definition both define a `:zone` with the same name, then the zone name will refer to the intersection of the two sets of constraints – even within formulas defined as part of layout marginalia.

6.1.3 Static Predicates

The simplest predicate is just a field name appropriate to the axis associated with the predicate. It selects very specifically that one sub-layout. To support `:frame` layouts with a "default" field, the underscore (_) is a valid name in this context.

You can supply a list of alternatives, separated by `|` vertical bar characters. In this case, each alternative is selected individually.

You can specify "all sub-fields *except* one or more alternatives" by prepending a `^` caret to the alternatives.

You may wish to specify merely that a particular axis has some value defined at this point. In that case, the `*` asterisk stands in for the set of all values. This is especially suited to certain applications of `:merge` patch-instructions and `:tree` layouts.

6.1.4 Computed Predicates

You can delegate a selection process to the host-language integration layer. For example, `@interesting` might implement a test for interesting games, so in context you could write `game=@interesting` as a criterion. In place of the word "interesting" you can substitute any identifier: the syntax is an `@`-sigil with base-name properly defined in your integration layer.

The implementation details are described in the integration chapter.

6.1.5 Selector Caveats

It is considered an error to constrain the same-named axis twice in a single selector. This is true even if one of the constraints is implied in a zone reference.

CHAPTER 7

Template Strings

Cubicle uses `"Double Quotes"` to delimit *template strings*. They can contain:

7.1 Replacement Parameters

Inside square brackets, put the name of an axis or any of several related forms. Here it is by example:

`"Subtotal [region] Sales"` At each cell where the template applies, the substring `[region]` gets replaced by the applicable value of the `region` axis, correctly mapped to plain text using the runtime-environment object.

`"[@foo]"` The `@` sigil means to insert the "raw" form of the current `foo` ordinal. If the replacement parameter is the *only* thing in the cell, then *cubicle* will use the native (e.g. numeric or date) form if possible.

`"Report for [.project] Sales"` This will ask the environment for a "global" parameter called `project` and substitute that in. This allows you to have punch-in parameters for the overall report rather than needing

`"Subtotal [foo.bar] Sales"` The environment is consulted to get the `bar` view of the current `foo` ordinal.

`"[case!1]"` An axis-header reference is particularly useful in merge-cell instructions, but may also find use elsewhere. The content is the (here, first) header associated with the current `case` ordinal. (Use a `2` for a second header, etc.)

Caveat: Any mentioned axis is assumed to exist in the address of any cell where the template is used. In the first example, if the template applies to a cell without a `region`, it will result in some sort of RUN-TIME error condition.

AT THE MOMENT, RUN-TIME ERRORS ARE NOT PRETTY.

The Future: I'd like to improve the handling of run-time layout errors, and also improve the advance validation, so that finding and fixing mistakes becomes easier.

7.2 Character Escapes and Line Breaks

The usual C-style *backslash-letter* escape codes (`abtnvfr`) are supported, although I can't imagine any use for these except for `\n`, for newline.

The aesthetics of that are dubious at best.

In general: You're going to want to break lines between words. The first word on the next line will generally be capitalized. Doing it with `\n` will be ugly and hard to read, especially for nontechnical people who might contribute copy.

Ugly Example: `"Multi-Line\nTitle Text"`

Therefore: Backslash appearing before a capital letter *becomes* a line-break, leaving the capital letter intact on the subsequent line.

Less-Ugly Example: `"Multi-Line\Title Text"`

Finally, you can use `\[`, `\\` and `\"` to represent a literal left-square-bracket, backslash, or double-quote, respectively. (Backslash before any other character is considered a syntax error.)

> If you provide a module definition as a triple-quoted string, it will be an excellent idea to make that string "raw", as in `r"""... \X ..."""`, to avoid quadruple-backslash heck.

CHAPTER 8

Formula Strings

Begin a formula with `@'` and finish it off with `'`. Leave out any leading =. *Cubicle* will supply that part for you.

Example: `@'if(1+1=2, "Good!", "Oops! Wrong Universe.")'`

8.1 Symbolic References

Formulas can contain symbolic cell references, as mentioned in the section on selectors. There are two types, illustrated by the following two *equivalent* examples: the `:` just inside the square brackets turns *off* the automatic summation feature. (Experience has shown sums are the most common whenever more than one cell is selected here.)

Equivalent Examples:

```
@'sum([:this=that,that=the_other])'
@'[this=that,that=the_other]'
```

Please note: Excel uses double-quotes to delimit literal strings within formulas.

It therefore makes sense that within *Cubicle* formula strings, double-quotes delimit *template strings* which get interpolated as such. Why? Because it's useful! Besides, when are you ever going to include a cell reference inside a literal string?

Open Issue: Currently, discontiguous selectors render as a comma-separated list of regular (cell or range) references. That is fine for taking sums, but can screw up the use of other formulas. It's not clear whether this is an actual problem.

CHAPTER 9

Integrating with Cubicle

This document shows you how to tie all the bits together and generate full-featured reports using a minimum of code.

9.1 Quick Start

Here's a minimal complete report generation program:

```
import xlsxwriter, os
from cubicle import compiler, dynamic, runtime, utility

module = compiler.compile_path("path/to/quickstart.cubicle")
env = runtime.Environment()
canvas = dynamic.Canvas(module, 'example', env)

for point, value in data_source(): # You define data_source().
        canvas.incr(point, value)

with xlsxwriter.Workbook('quickstart.xlsx') as book:
        sheet = book.add_worksheet()
        canvas.plot(book, sheet, 0, 0, 'blank')

utility.startfile('quickstart.xlsx') # Don't get me started...
```

What is going on here?

- Import the bits you need.
- Build the cubicle module corresponding to the report you want to format.
- Supply a "runtime environment" which connects your business layer.
- Instantate a `Canvas` object.
- Fill data into said `Canvas` object.
- Plot the canvas into a suitable workbook/worksheet pair.

- Open the resulting file for the end-user.

You can learn the `cubicle` language from the *language* chapter. You can substitute `compile_string(...)` if you prefer your report definition inline with the report program, although if you have a sizable suite of reports you maintain, you probably want to put them all in a common external file and pull out the specific canvas you need.

9.2 Supplying Data

At the moment, there are three methods considered as part of the public API for supplying data to fill in a report. They all have a common signature: each method expects a `point` and a `value`.

As used in the API, `point` parameters are just dictionaries. You fill in the keys in such manner as to indicate a distinct cell according to the layout structure for your canvas.

9.2.1 One-at-a-time Operations

`canvas.incr(point, value)` This is probably your most commonly used method. It adds the supplied (numeric) value to the value already stored at the layout cell addressed by the supplied point. If no such value exists yet, the starting value is zero. Also, any `:tree` or `:menu` along the way will automatically create any necessary children to make sure that an appropriate cell exists

In the unlikely event you supply an ordinal for a `:frame` or `:menu` element which does not match a known field, this is considered a bug in the caller and some sort of exception will be tossed in your general direction.

`canvas.decr(point, value)` This is equivalent to `canvas.incr(point, 0-value)` but may express intent a bit more clearly: a decrement rather than an increment.

`canvas.poke(point, value)` This sets or replaces the value currently in the cell addressed by the supplied point. You can use any value type which `xlsxwriter` supports writing out to a spreadsheet: strings, numbers, dates/times, even URL objects. If you `.poke(...)` a value which cannot be incremented (or decremented) in place, then do please apply common sense with respect to the `.incr(...)` and `.decr(...)` methods.

9.2.2 Data Stream Operations

Use the `for`s, Luke.

9.2.3 Using Named Zones

Once you've created a `dynamic.Canvas` object, you can ask it for a dictionary which represents a defined zone as known to its layout structures.

```
canvas = dynamic.Canvas(module, 'example', env)
apples = canvas.zone('apples')
oranges = canvas.zone('oranges')

... and then later ...

canvas.incr({**apples, **point}, apple_value)
canvas.incr({**oranges, **point}, orange_value)
```

This gives you the freedom to re-jigger your layout cosmetically as long as the named-zones expose the correct semantics.

Computed-Axis Caveat At least for now, a named zone inside of a computed-axis `:frame` or `:menu` structure can be used for boilerplate and formatting, but it probably makes no sense to refer to such a zone from the application, because it means supplying an ordinal which would have been computed anyway. Perhaps one day that won't be valid? For the meantime, I would not rely on such behavior.

9.3 Business Logic and Domain Knowledge

You'll normally extend `runtime.Environment` class and supply your own instance instead of using the completely generic version. It comes pre-built with some bits to simplify plugging predicates, collations, and inferences appropriate to your application domain.

```
class MyEnv(runtime.Environment):
        ... Application-specific customization goes here ...

... and then later ...

env = MyEnv()
canvas = dynamic.Canvas(module, 'example', env)
```

The interface between the `dynamic.Canvas` class and the `runtime.Environment` class is pretty close to its final form: it might gain another method or parameter, but the basic design seems sound enough, so you should be safe to experiment with alternative implementations.

The present *default implementations* of those interface methods provide the API described below, which *MAY BE* subject to at least some change.

Development Note: Currently this section is in DTSTTCPW mode, but as patterns of use and limitations become apparent, some adjustments are scheduled for version 0.9.0. In particular, collation often goes hand-in-hand with making data fit for people to gaze upon (the "friendly-name" problem).

9.3.1 Computed Predicates

You can implement a method like this:

```
class MyEnv(runtime.Environment):
        ...
        def is_interesting(self, game: str):
                return game.startswith('Benko')
        ...
```

With that in place, you can use `game=@interesting` anywhere a field predicate is called for in the cubicle module.

Open question: Should the axis name be passed in?

9.3.2 Computed Axes (e.g. Default Categories)

Suppose you report on groceries, and you frequently group them by "produce / meats / dry-goods" categorization. Maybe you call that "department". So most of your data sources will supply a food ID, and most of your reports need to know the department. You don't want to have to modify the data sources. Instead, make your reports use a computed axis `@department`, and then implement as follows:

```
FOOD_DEPARTMENT = {...}  # Maybe query a database ahead of time.

class MyEnv(runtime.Environment):
        ...
        def magic_department(self, point:dict):
                food = point['food']
                return FOOD_DEPARTMENT[food]
        ...
```

Now any time a report has a `:tree`, `:frame`, or `:menu` with the axis specified as `@department` instead of `department`, then Project Cubicle will consult this method instead of expecting to find the department passed along in the data stream.

Why the `magic_` prefix? No reason. It's magic.

9.3.3 Custom Collation

Going back to the groceries example, perhaps you've got a dozen departments with a conventional order in which these should always appear within reports, but you don't want to spell this out explicitly all over the place. In that case:

```
class MyEnv(runtime.Environment):
        ...
        def collate_department(self, department):
                return ... a comparison key ...
        ...
```

Now when you use `... :tree department ...` (or `... :tree @department ...`) in your cubicle definition, the layout will respect the collation order you've defined here.

9.3.4 "Friendly Names"

Consider again the groceries. Everything in the store has a SKU number. (That's "stock-keeping unit" for the uninitiated.) Everything in the store's database is keyed to these numbers. But nobody thinks of SKU #1405. Unless you've been working the check stands all summer, you think of red bell peppers.

See also https://en.wiktionary.org/wiki/friendly_name

We'd like to be able to hand a SKU number to the canvas and know that, in presentation, it will appear in plain English. Except that sometimes, you actually do need to see the SKU.

This part isn't mature yet, but in concept the runtime environment object you supply should also facilitate this kind of idea.

For the moment, you can override the `.plain_text(...)` method, perhaps to grub around for specially-named methods, but longer-term, the plan is to make something a bit nicer.

CHAPTER 10

Known Issues and Bugs

- The automatic summation feature could be smarter and only wrap `sum(...)` around references to two or more cells.
- Those error messages not related to parsing aren't always too helpful.

If anything is unclear, please feel free to file an issue or contact me through GitHub. Feedback is always welcome.

CHAPTER 11

Indices and tables

- genindex
- search