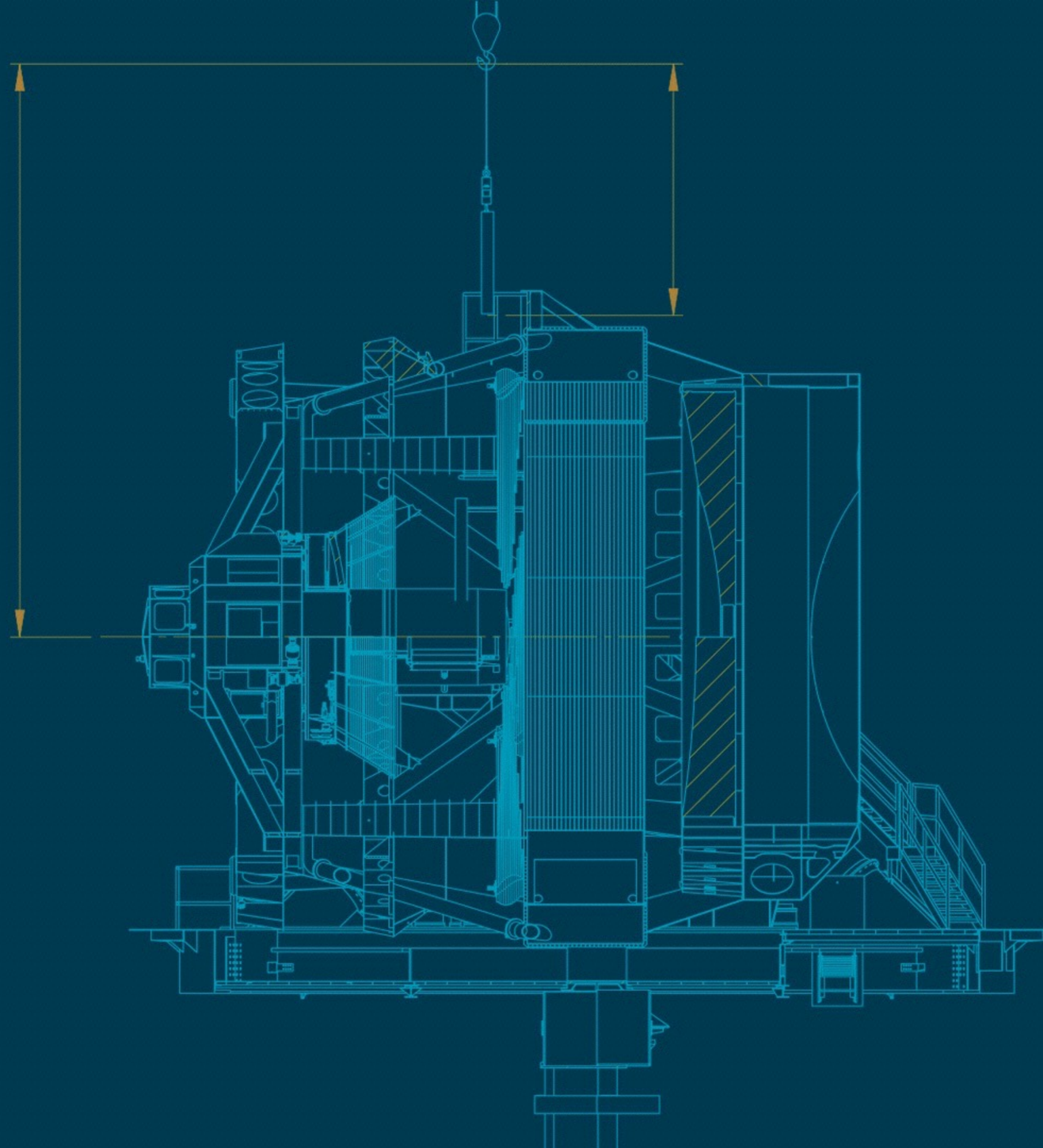
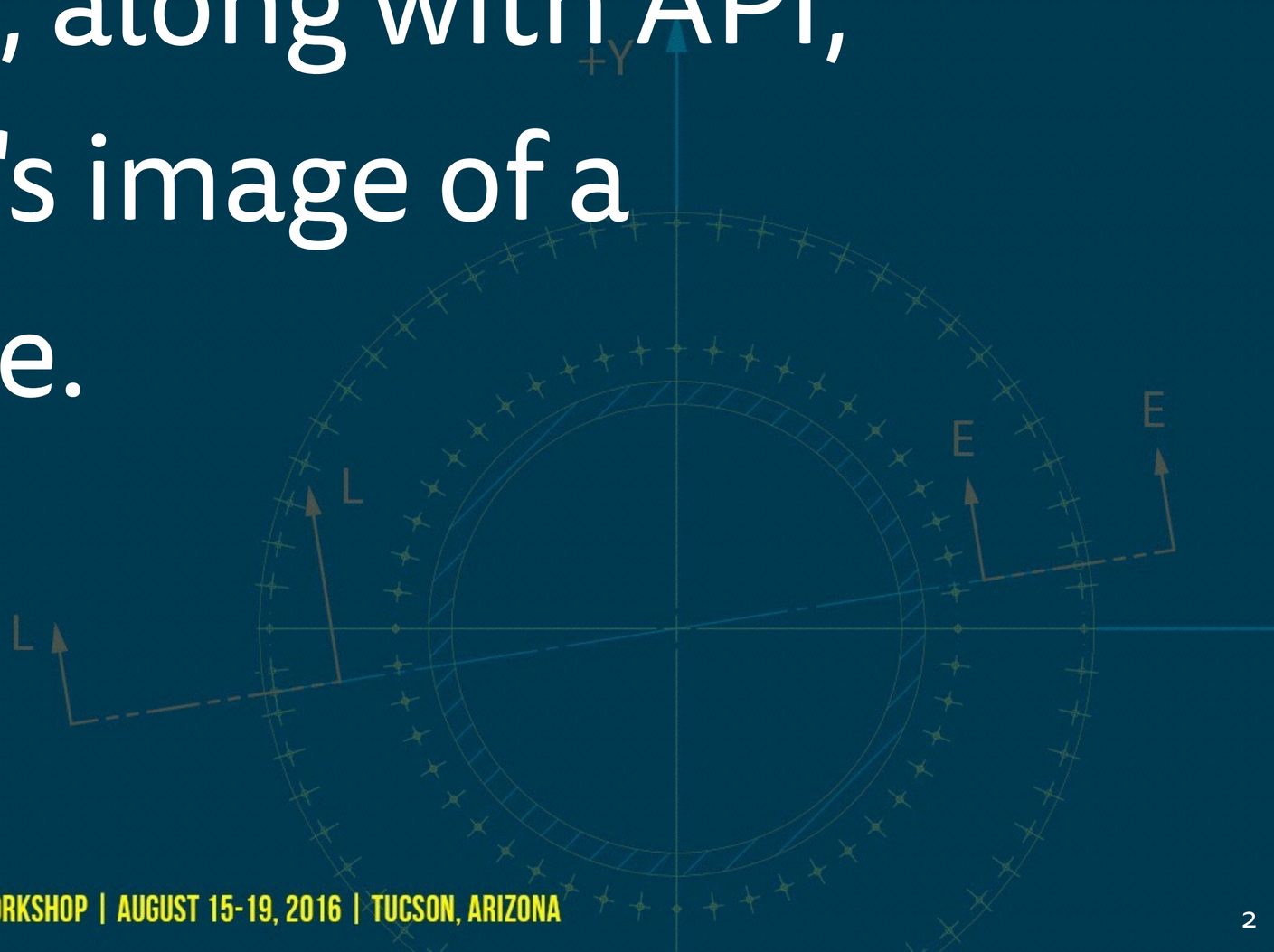
Let's Talk About Docs

Jonathan Sick
SQuaRE



LSST2016 PROJECT AND COMMUNITY WORKSHOP
AUGUST 15-19, 2016 | TUCSON, ARIZONA

Documentation, along with API, frames the user's image of a piece of software.



Documentation is the best marketing.

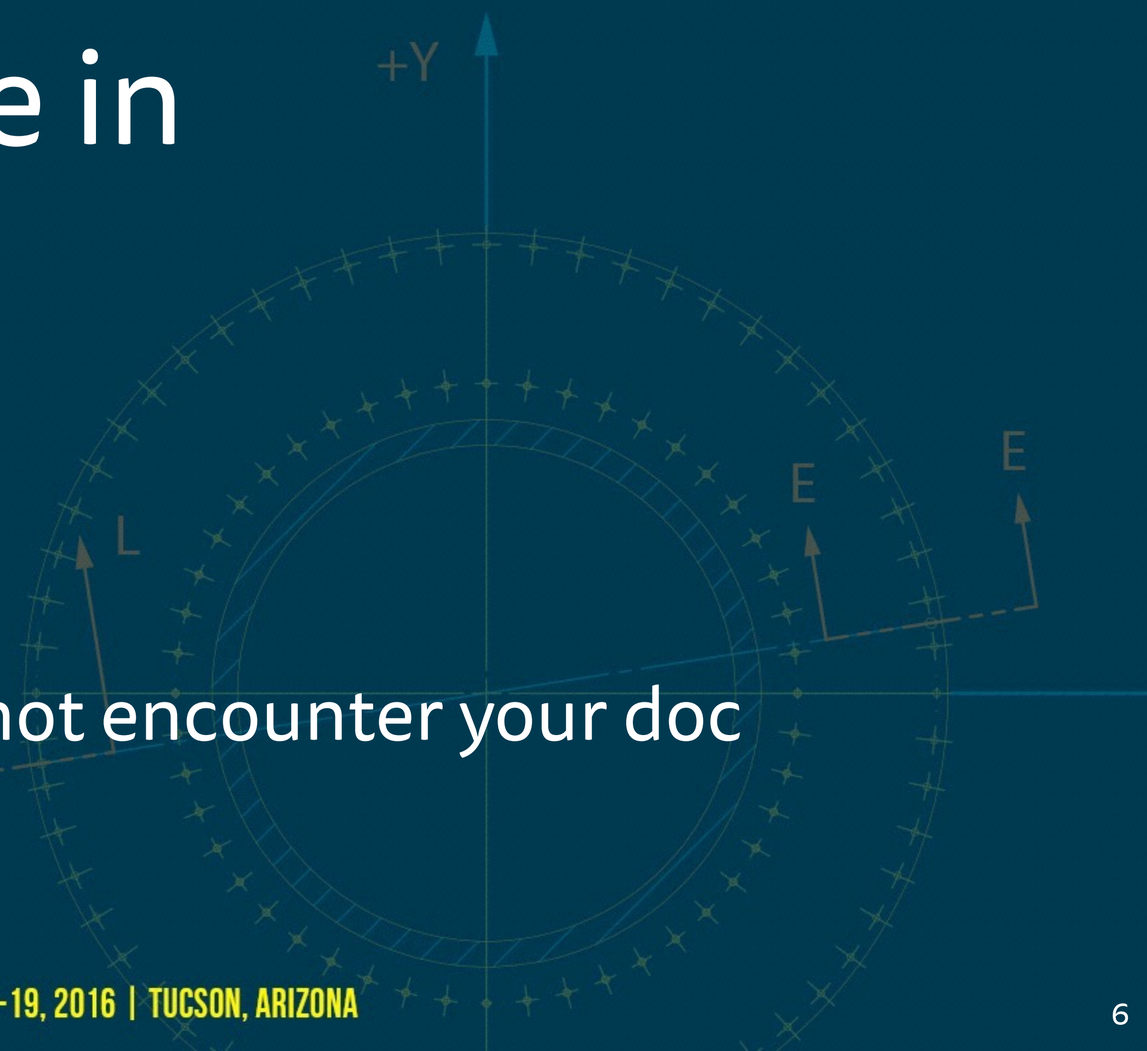
Docs are first class products of software development.

Docs or it didn't happen.

Writing is an exercise in empathy.

You’re writing for the reader.

Remember the reader might not encounter your doc in the best circumstances.



What is not documentation?

Source code.

Unit tests.

Types of Documentation

Tutorial

Experience.

User Guide

Understanding.

Reference

Reliable companion.

Tutorials

- Give a realistic feeling for using the software on real projects
  - Conceptual docs for concepts & 'toy' examples; tutorials for real-world experience
- Demonstrate best practices
- Keep explanations light; link liberally to reference & concept docs
- 30 minutes or less: lunch break sized
  - Chain tutorials to do bigger, more complex things
- Railroad the reader towards a goal or product
  - Show how to arrive at solution, at every step. 'Exercise for the reader' doesn't cut it.
  - Show solution. Ideally we'd like to provide downloadable repos or notebooks.
- State pre-requisites up-front

THE COURSE MODULES

Each video breaks down a specific part of React and allows for quick referencing in the future.

Just under 5 hours of video — learn React in an afternoon or two!

DEVELOPMENT ENVIRONMENT SETUP WITH GULP + BROWSERIFY

INTRODUCTION TO REACT COMPONENTS

WRITING YOUR FIRST COMPONENT

WRITING HTML WITH JSX

CREATING OUR APPLICATION LAYOUT WITH COMPONENTS

PASSING DYNAMIC DATA WITH PROPS

ROUTING WITH REACT ROUTER

USING UTILITY : HELPER FUNCTIONS IN REACT

ALL ABOUT REACT EVENT LISTENERS

UNDERSTANDING STATE

LOADING DATA INTO STATE ONCLICK

DISPLAYING STATE WITH JSX

UPDATING OUR ORDER STATE

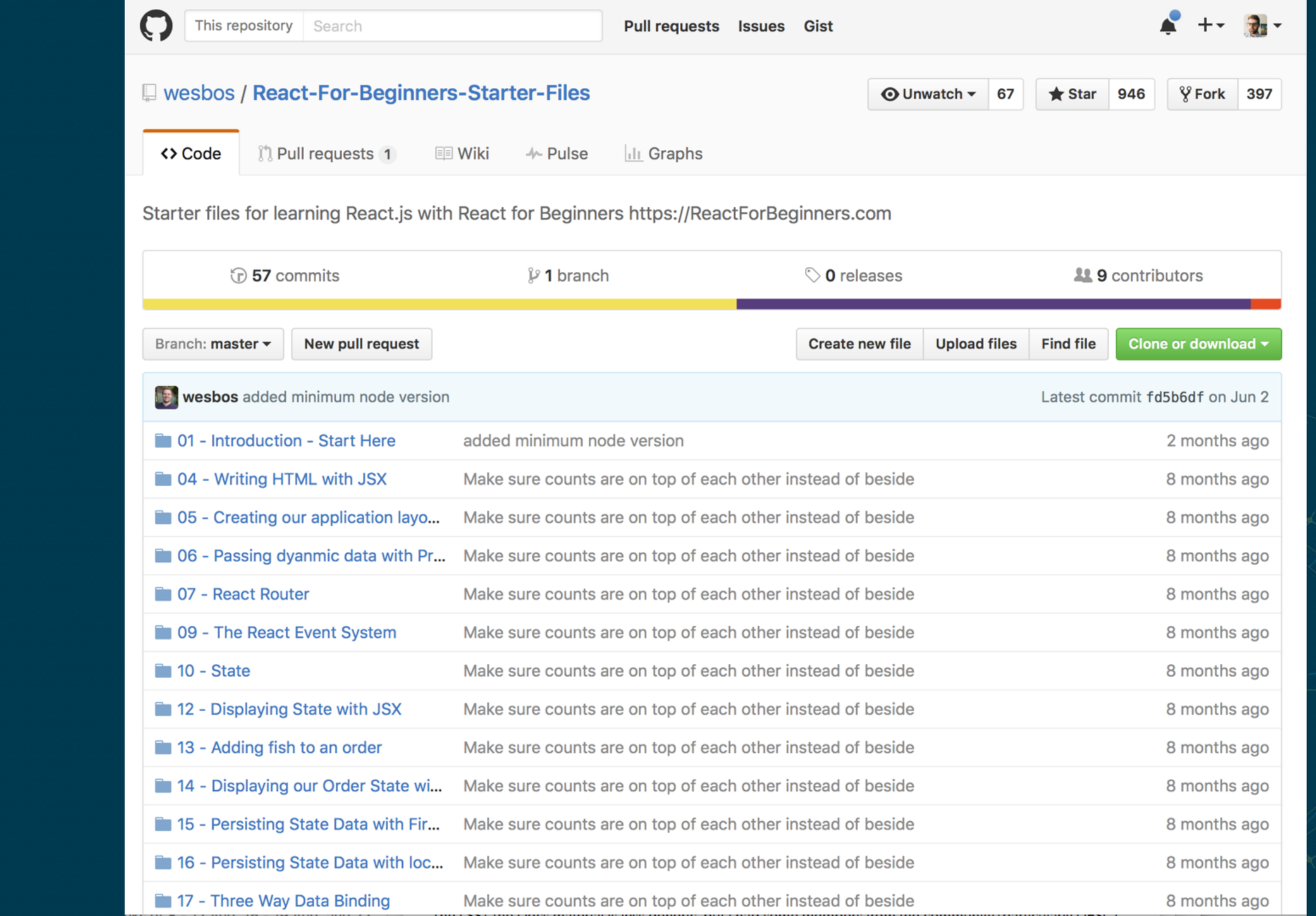
DISPLAYING OUR ORDER STATE WITH JSX

PERSISTING STATE WITH FIREBASE

PERSISTING STATE WITH LOCALSTORAGE

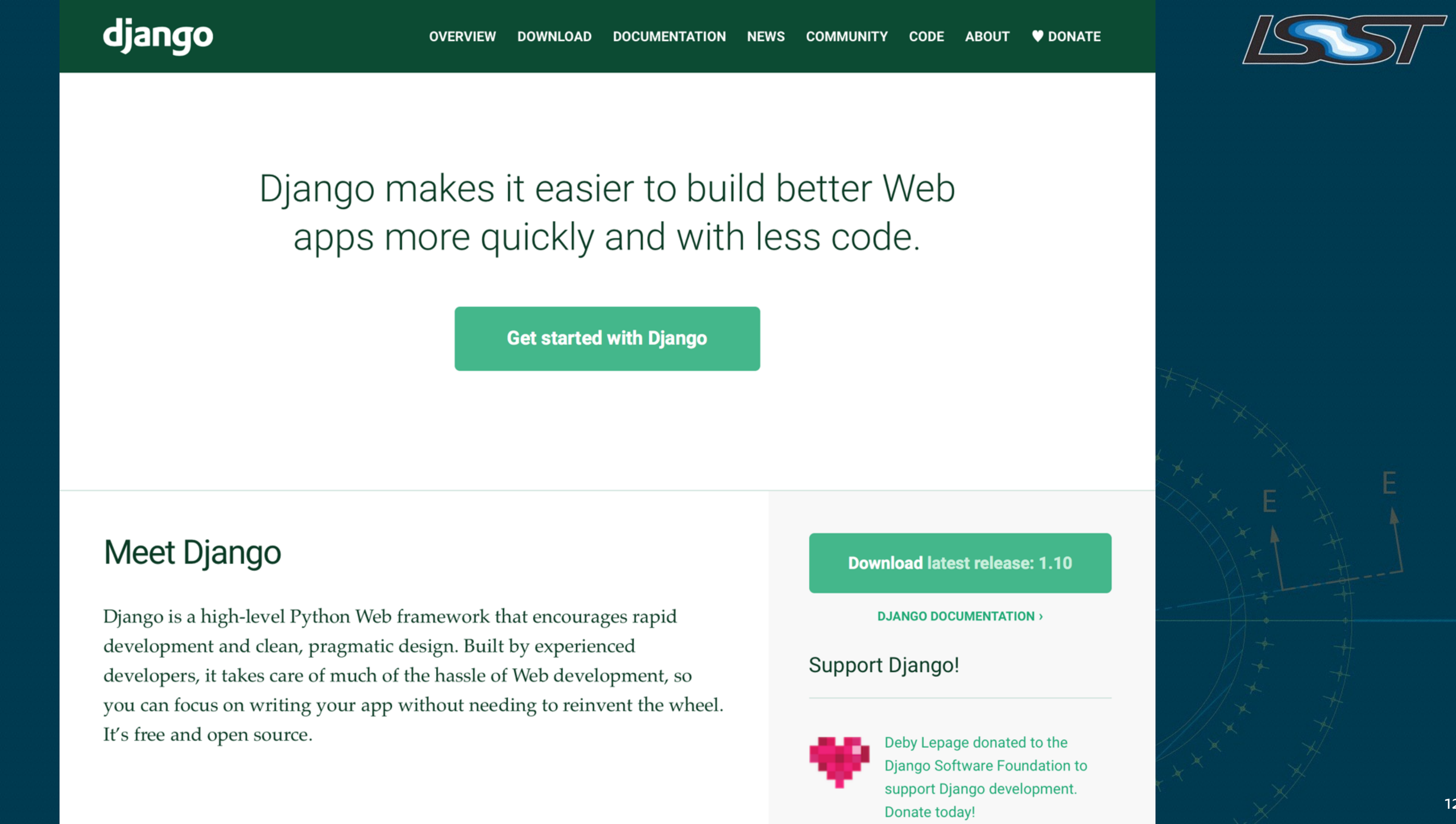
BI-DIRECTIONAL DATA FLOW WITH LINKSTATE

UPDATING AND DELETING STATE



Realistic

Railroading


django
OVERVIEW
DOWNLOAD
DOCUMENTATION
NEWS
COMMUNITY
CODE
ABOUT
♥ DONATE
Django makes it easier to build better Web apps more quickly and with less code.
Get started with Django
Meet Django
Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of Web development, so you can focus on writing your app without needing to reinvent the wheel. It's free and open source.
Download latest release: 1.10
DJANGO DOCUMENTATION ›
Support Django!
Deby Lepage donated to the Django Software Foundation to support Django development. Donate today!

Getting started with Django

Depending how new you are to Django, you can try a tutorial, or just dive into the documentation.

Want to learn more about Django? Read the overview to see whether Django is right for your project.

DJANGO OVERVIEW ›

Install Django

Before you can use Django, you'll need to install it. Our complete installation guide covers all the possibilities; this guide will get you to a simple, minimal installation that'll work while you walk through the introduction.

DJANGO INSTALLATION GUIDE ›

Write your first Django app

Writing your first Django app, part 1

Let's learn by example.

Throughout this tutorial, we'll walk you through the creation of a basic poll application.

It'll consist of two parts:

- A public site that lets people view polls and vote in them.
- An admin site that lets you add, change, and delete polls.

We'll assume you have Django installed already. You can tell Django is installed and which version by running the following command:

```
$ python -m django --version
```

If Django is installed, you should see the version of your installation. If it isn't, you'll get an error telling "No module named django".

This tutorial is written for Django 1.10 and Python 3.4 or later. If the Django version doesn't match, you can refer to the tutorial for your version of Django by using the version switcher at the bottom right corner of this page, or update Django to the newest version. If you are still using Python 2.7, you will need to adjust the code samples slightly, as described in comments.

See How to install Django for advice on how to remove older versions of Django and install a newer one.

Where to get help:

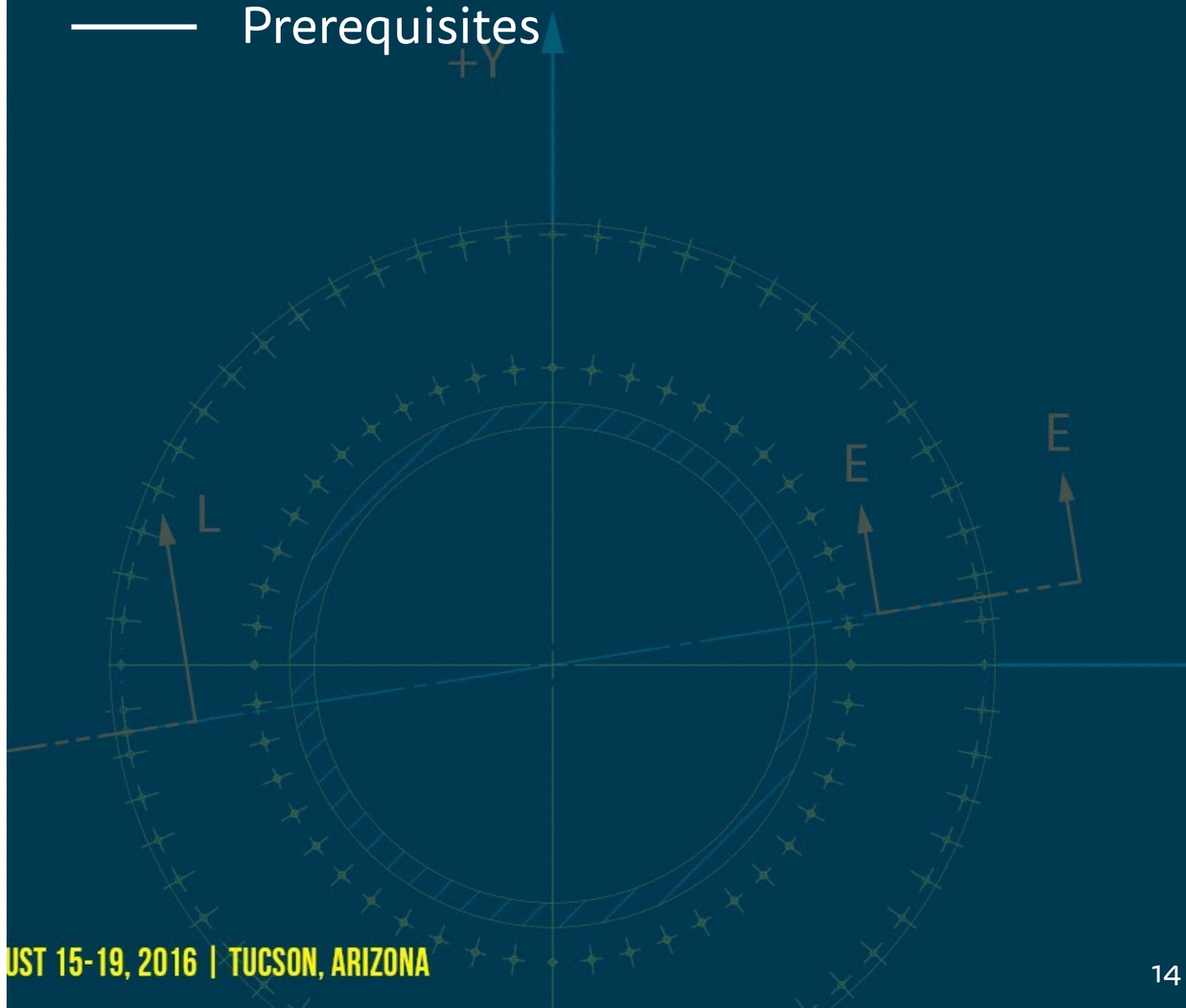
If you're having trouble going through this tutorial, please post a message to django-users or drop by #django on irc.freenode.net to chat with other Django users who might be able to help.

Creating a project

If this is your first time using Django, you'll have to take care of some initial setup. Namely, you'll need to auto-generate some code that establishes a Django project – a collection of settings

— Goals / end result

— Prerequisites



Write your first view

Let's write the first view. Open the file **polls/views.py** and put the following Python code in it:

polls/views.py

```
from django.http import HttpResponse


def index(request):
    return HttpResponse("Hello, world. You're at the polls
index.")
```

This is the simplest view possible in Django. To call the view, we need to map it to a URL - and for this we need a URLconf.

To create a URLconf in the polls directory, create a file called **urls.py**. Your app directory should now look like:

```
polls/
    __init__.py
    admin.py
    apps.py
    migrations/
        __init__.py
    models.py
    tests.py
    urls.py
    views.py
```

In the **polls/urls.py** file include the following code:

polls/urls.py

```
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
]
```

Short paragraphs;
clear instructions.

Checkpoints help a
reader follow along.

First steps

Are you new to Django or to programming? This is the place to start!

- **From scratch:** Overview | Installation
- **Tutorial:** Part 1: Requests and responses | Part 2: Models and the admin site | Part 3: Views and templates | Part 4: Forms and generic views | Part 5: Testing | Part 6: Static files | Part 7: Customizing the admin site
- **Advanced Tutorials:** How to write reusable apps | Writing your first patch for Django

Conceptual Docs / User Guide

- This is about teaching
  - Think about the project's philosophy and teach that
  - Use clear, simple words (more on writing later)
- Be comprehensive
  - More details than tutorials, but still link frequently to reference
- Use illustrative examples
  - Examples can be stripped-down, but should work
  - Names should be relevant; avoid foo and bar

Page Contents

Data Tables (`astropy.table`)

Introduction

`astropy.table` provides functionality for storing and manipulating heterogeneous tables of data in a way that is familiar to `numpy` users. A few notable capabilities of this package are:

- Initialize a table from a wide variety of input data structures and types.
- Modify a table by adding or removing columns, changing column names, or adding new rows of data.
- Handle tables containing missing values.
- Include table and column metadata as flexible data structures.
- Specify a description, units and output formatting for columns.
- Interactively scroll through long tables similar to using `more`.
- Create a new table by selecting rows or columns from a table.
- Perform Table operations like database joins, concatenation, and binning.
- Maintain a table index for fast retrieval of table items or ranges.
- Manipulate multidimensional columns.
- Handle non-native (mixin) column types within table.
- Methods for Reading and writing Table objects to files.
- Hooks for Subclassing Table and its component classes.

Currently `astropy.table` is used when reading an ASCII table using `astropy.io.ascii`. Future releases of AstroPy are expected to use the `Table` class for other subpackages such as `astropy.io.votable` and `astropy.io.fits`.

Getting Started

The basic workflow for creating a table, accessing table elements, and modifying the table is shown below. These examples show a very simple case, while the full `astropy.table` documentation is available from the Using table section.

First create a simple table with three columns of data named `a`, `b`, and `c`. These columns have integer, float, and string values respectively:

```
>>> from astropy.table import Table
>>> a = [1, 4, 5]
>>> b = [2.0, 5.0, 8.2]
>>> c = ['x', 'y', 'z']
>>> t = Table([a, b, c], names=('a', 'b', 'c'), meta={'name': 'first table'})
```

If you have row-oriented input data such as a list of records, use the `rows` keyword. In this example we also explicitly set the data types for each column:

```
>>> data_rows = [(1, 2.0, 'x'),
...              (4, 5.0, 'y'),
...              (5, 8.2, 'z')]
>>> t = Table(rows=data_rows, names=('a', 'b', 'c'), meta={'name': 'first table'},
...           dtype=('i4', 'f8', 'S1'))
```

There are a few ways to examine the table. You can get detailed information about the table values and column definitions as follows:

v: stable

1. Introduction
2. Getting Started
3. Using {subpackage}
4. Reference/API

Quickly and clearly answer the question:

What is this? Will this solve my problem?

ARIZONA

Getting Started

The basic workflow for creating a table, accessing table elements, and modifying the table is shown below. These examples show a very simple case, while the full **`astropy.table`** documentation is available from the Using table section.

First create a simple table with three columns of data named `a`, `b`, and `c`. These columns have integer, float, and string values respectively:

```
>>> from astropy.table import Table
>>> a = [1, 4, 5]
>>> b = [2.0, 5.0, 8.2]
>>> c = ['x', 'y', 'z']
>>> t = Table([a, b, c], names=('a', 'b', 'c'), meta={'name': 'first table'})
```

If you have row-oriented input data such as a list of records, use the `rows` keyword. In this example we also explicitly set the data types for each column:

```
>>> data_rows = [(1, 2.0, 'x'),
...              (4, 5.0, 'y'),
...              (5, 8.2, 'z')]
>>> t = Table(rows=data_rows, names=('a', 'b', 'c'), meta={'name': 'first table'},
...           dtype=('i4', 'f8', 'S1'))
```

There are a few ways to examine the table. You can get detailed information about the table values and column definitions as follows:

```
>>> t
<Table length=3>
  a      b     c
int32 float64 str1
----- ------- ----
    1     2.0    x
    4     5.0    y
    5     8.2    z
```

You can also assign a unit to the columns. If any column has a unit assigned, all units would be shown as follows:

```
>>> t['b'].unit = 's'
>>> t
<Table length=3>
  a      b       c
         s
int32 float64 str1
----- ------- ----
    1     2.0    x
    4     5.0    y
    5     8.2    z
```

Finally, you can get summary information about the table as follows:

1. Introduction
2. Getting Started
3. Using {subpackage}
4. Reference/API

Show what it's like to use API with small examples.

Like a lightweight tutorial.

Using `table`

The details of using **`astropy.table`** are provided in the following sections:

Construct table

- Constructing a table
  - Examples
  - Initialization Details
  - Copy versus Reference
  - Column and TableColumns classes
  - Subclassing Table
  - Table-like objects

Access table

- Accessing a table
  - Quick overview
  - Details

Modify table

- Modifying a table
  - Quick overview
  - Caveats

Table operations ¶

- Table operations
  - Grouped operations
  - Binning
  - Stack vertically
  - Stack horizontally
  - Join
  - Merging details
  - Unique rows

Indexing

- Table indexing
  - Creating an index
  - Row retrieval using indices
  - Effects on performance
  - Index modes
  - Engines

Masking

- Masking and missing values
  - Table creation
  - Table access
  - Masking and filling

1. Introduction
2. Getting Started
3. Using {subpackage}
4. Reference/API

Detailed, per-concept guides.

Page Contents

Table operations

In this section we describe higher-level operations that can be used to generate a new table from one or more input tables. This includes:

Documentation	Description	Function
Grouped operations	Group tables and columns by keys	`group_by`
Binning	Binning tables	`group_by`
Stack vertically	Concatenate input tables along rows	`vstack`
Stack horizontally	Concatenate input tables along columns	`hstack`
Join	Database-style join of two tables	`join`
Unique rows	Unique table rows by keys	`unique`

Grouped operations

Sometimes in a table or table column there are natural groups within the dataset for which it makes sense to compute some derived values. A simple example is a list of objects with photometry from various observing runs:

```
>>> from astropy.table import Table
>>> obs = Table.read("""name    obs_date    mag_b  mag_v
...                     M31     2012-01-02  17.0   17.5
...                     M31     2012-01-02  17.1   17.4
...                     M101    2012-01-02  15.1   13.5
...                     M82     2012-02-14  16.2   14.5
...                     M31     2012-02-14  16.9   17.3
...                     M82     2012-02-14  15.2   15.5
...                     M101    2012-02-14  15.0   13.6
...                     M82     2012-03-26  15.7   16.5
...                     M101    2012-03-26  15.1   13.5
...                     M101    2012-03-26  14.8   14.3
...                     """, format='ascii')
```

Table groups

Now suppose we want the mean magnitudes for each object. We first group the data by the `name` column with the `group_by()` method. This returns a new table sorted by `name` which has a `groups` property specifying the unique values of `name` and the corresponding table rows:

```
>>> obs_by_name = obs.group_by('name')
>>> print(obs_by_name)
name  obs_date  mag_b mag_v
---- ---------- ----- -----
M101 2012-01-02  15.1  13.5  << First group (index=0, key='M101')
M101 2012-02-14  15.0  13.6
M101 2012-03-26  15.1  13.5
M101 2012-03-26  14.8  14.3
 M31 2012-01-02  17.0  17.5  << Second group (index=4, key='M31')
```

v: stable

1. Introduction
2. Getting Started
3. Using {subpackage}
4. Reference/API

Lots of illustrative examples.

Getting started with Django

Depending how new you are to Django, you can try a tutorial, or just dive into the documentation.

Want to learn more about Django? Read the overview to see whether Django is right for your project.

DJANGO OVERVIEW ›

Install Django

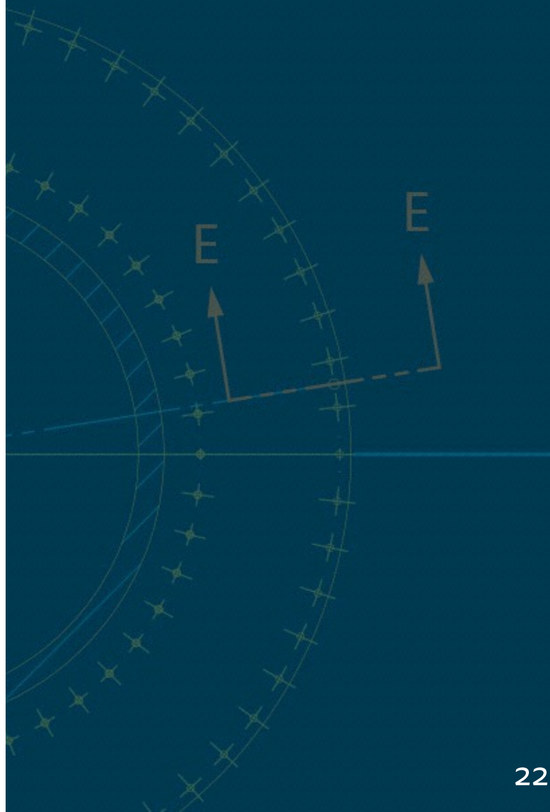
Before you can use Django, you'll need to install it. Our complete installation guide covers all the possibilities; this guide will get you to a simple, minimal installation that'll work while you walk through the introduction.

DJANGO INSTALLATION GUIDE ›

Write your first Django app

Installed Django already? Good. Now try this tutorial, which walks you through creating a

Revisiting Django.



Intro to Django

EXPAND ALL / COLLAPSE ALL

Object-relational mapper

Define your data models entirely in Python. You get a rich, dynamic database-access API for free — but you can still write SQL if needed.

READ MORE ›

```python
class Band(models.Model):
    """A model of a rock band."""
    name = models.CharField(max_length=200)
    can_rock = models.BooleanField(default=True)


class Member(models.Model):
    """A model of a rock band member."""
    name = models.CharField("Member's name", max_length=200)
    instrument = models.CharField(choices=(
            ('g', "Guitar"),
            ('b', "Bass"),
            ('d', "Drums"),
        ),
        max_length=1
    )
    band = models.ForeignKey("Band")
```

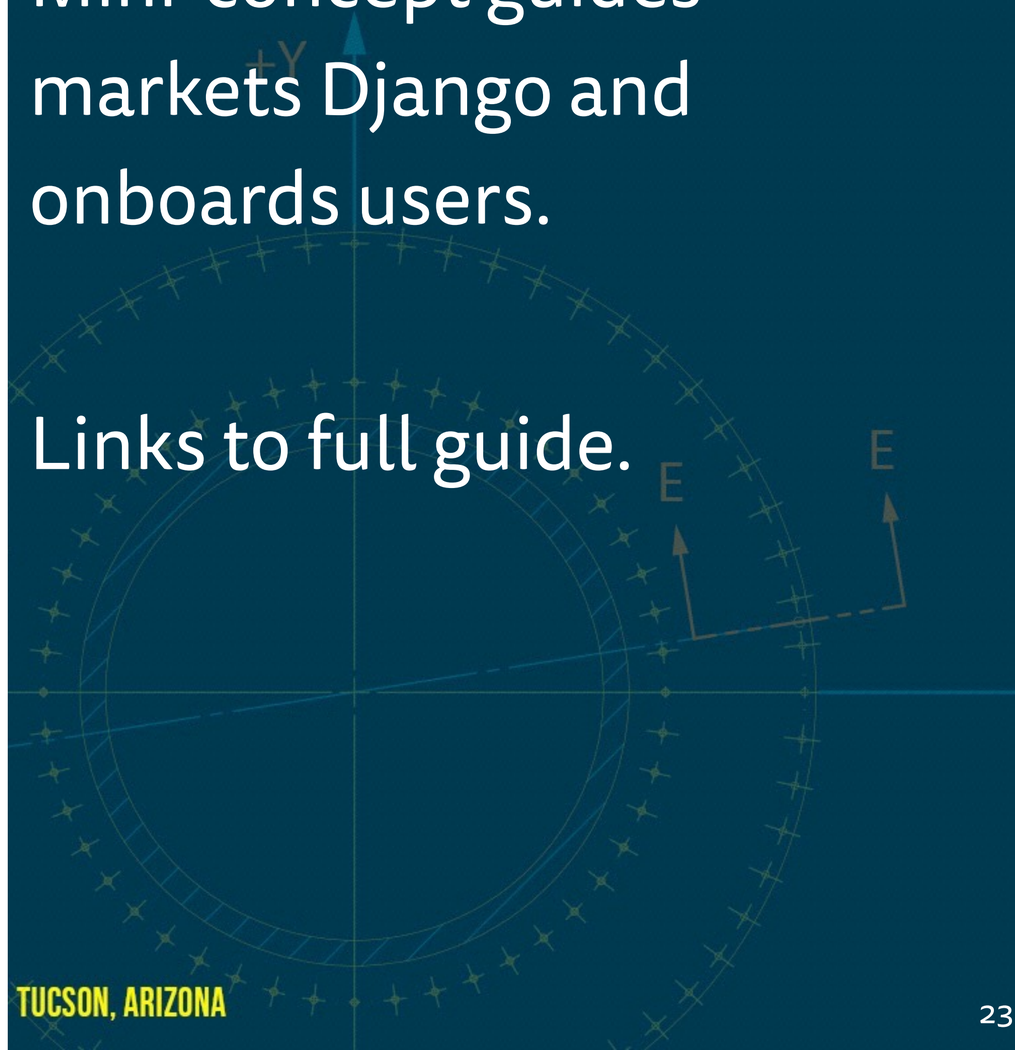
URLs and views

Templates

Forms

Mini-concept guides markets Django and onboards users.

Links to full guide.



TUCSON, ARIZONA

Models

A model is the single, definitive source of information about your data. It contains the essential fields and behaviors of the data you're storing. Generally, each model maps to a single database table.

The basics:

- Each model is a Python class that subclasses **`django.db.models.Model`**.
- Each attribute of the model represents a database field.
- With all of this, Django gives you an automatically-generated database-access API; see Making queries.

Quick example

This example model defines a **`Person`**, which has a **`first_name`** and **`last_name`**:

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

`first_name` and **`last_name`** are fields of the model. Each field is specified as a class attribute, and each attribute maps to a database column.

The above **`Person`** model would create a database table like this:

```
CREATE TABLE myapp_person (
    "id" serial NOT NULL PRIMARY KEY,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(30) NOT NULL
);
```

Some technical notes:

- The name of the table, **`myapp_person`**, is automatically derived from some model metadata but can be overridden. See Table names for more details.
- An **`id`** field is added automatically, but this behavior can be overridden. See Automatic primary key fields.
- The **`CREATE TABLE`** SQL in this example is formatted using PostgreSQL syntax, but it's worth noting Django uses SQL tailored to the database backend specified in your settings file.

Using models

Once you have defined your models, you need to tell Django you're going to *use* those models. Do this by editing your settings file and changing the **`INSTALLED_APPS`** setting to add the name of the module that contains your **`models.py`**.

For example, if the models for your application live in the module **`myapp.models`** (the package structure that is created for an application by the **`manage.py startapp`** script), **`INSTALLED_APPS`** should read, in part:

```
INSTALLED_APPS = [
    #...
    'myapp',
    #...
]
```

When you add new apps to **`INSTALLED_APPS`**, be sure to run **`manage.py migrate`**, optionally making migrations for them first with **`manage.py makemigrations`**.

Fields

The most important part of a model – and the only required part of a model – is the list of database fields it defines. Fields are specified by class attributes. Be careful not to choose field names that conflict with the models API like **`clean`**, **`save`**, or **`delete`**.

Example:

```
from django.db import models

class Musician(models.Model):
```

Reference Guides

- API references extracted from code comments/docstrings
  - Formats: Numpydoc for Python, Doxygen for C++
- Used in many contexts: doc site, python help(), devs viewing code
- Strongly formalized format
  - One sentence summary
  - Parameters, returns, exceptions
  - Usage examples
  - Possibly also detailed notes, ‘see also,’ literature references for algorithms
- Separation between public API & internal implementation
  - ‘_objects’ are excluded; **use public import paths**
  - Use regular code comments for things an API user shouldn’t know

numpy.sin

numpy.sin(*x*[, *out*]) = <ufunc 'sin'>

Trigonometric sine, element-wise.

Parameters: **x** : *array_like*
Angle, in radians (2π rad equals 360 degrees).

Returns: **y** : *array_like*
The sine of each element of x.

See also:
arcsin, sinh, cos

Notes

The sine is one of the fundamental functions of trigonometry (the mathematical study of triangles). Consider a circle of radius 1 centered on the origin. A ray comes in from the $+x$ axis, makes an angle at the origin (measured counter-clockwise from that axis), and departs from the origin. The y coordinate of the outgoing ray's intersection with the unit circle is the sine of that angle. It ranges from -1 for $x = 3\pi/2$ to +1 for $\pi/2$. The function has zeroes where the angle is a multiple of π. Sines of angles between π and 2π are negative. The numerous properties of the sine and related functions are included in any standard trigonometry text.

Examples

Print sine of one angle:

```
>>> np.sin(np.pi/2.)
1.0
```

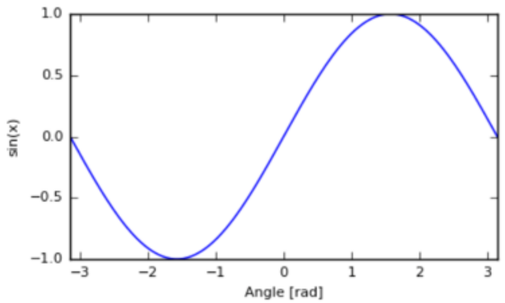
Print sines of an array of angles given in degrees:

```
>>> np.sin(np.array((0., 30., 45., 60., 90.)) * np.pi / 180. )
array([ 0.        ,  0.5       ,  0.70710678,  0.8660254 ,  1.        ])
```

Plot the sine function:

```
>>> import matplotlib.pylab as plt
>>> x = np.linspace(-np.pi, np.pi, 201)
>>> plt.plot(x, np.sin(x))
>>> plt.xlabel('Angle [rad]')
>>> plt.ylabel('sin(x)')
>>> plt.axis('tight')
>>> plt.show()
```

(Source code, png, pdf)



- Signature (auto)
- One-sentence summary.
- Parameter/Return Details
- Helpful links
- Detailed explanation of behaviour; things users should be aware of
- Examples of behaviours and usage ideas

Reference/API

astropy.table Package

Functions

hstack(tables[, join_type, uniq_col_name, ...])	Stack tables along columns (horizontally)
join(left, right[, keys, join_type, ...])	Perform a join of the left table with the right table on specified keys.
unique(input_table[, keys, silent, keep])	Returns the unique rows of a table.
vstack(tables[, join_type, metadata_conflicts])	Stack tables vertically (along rows)

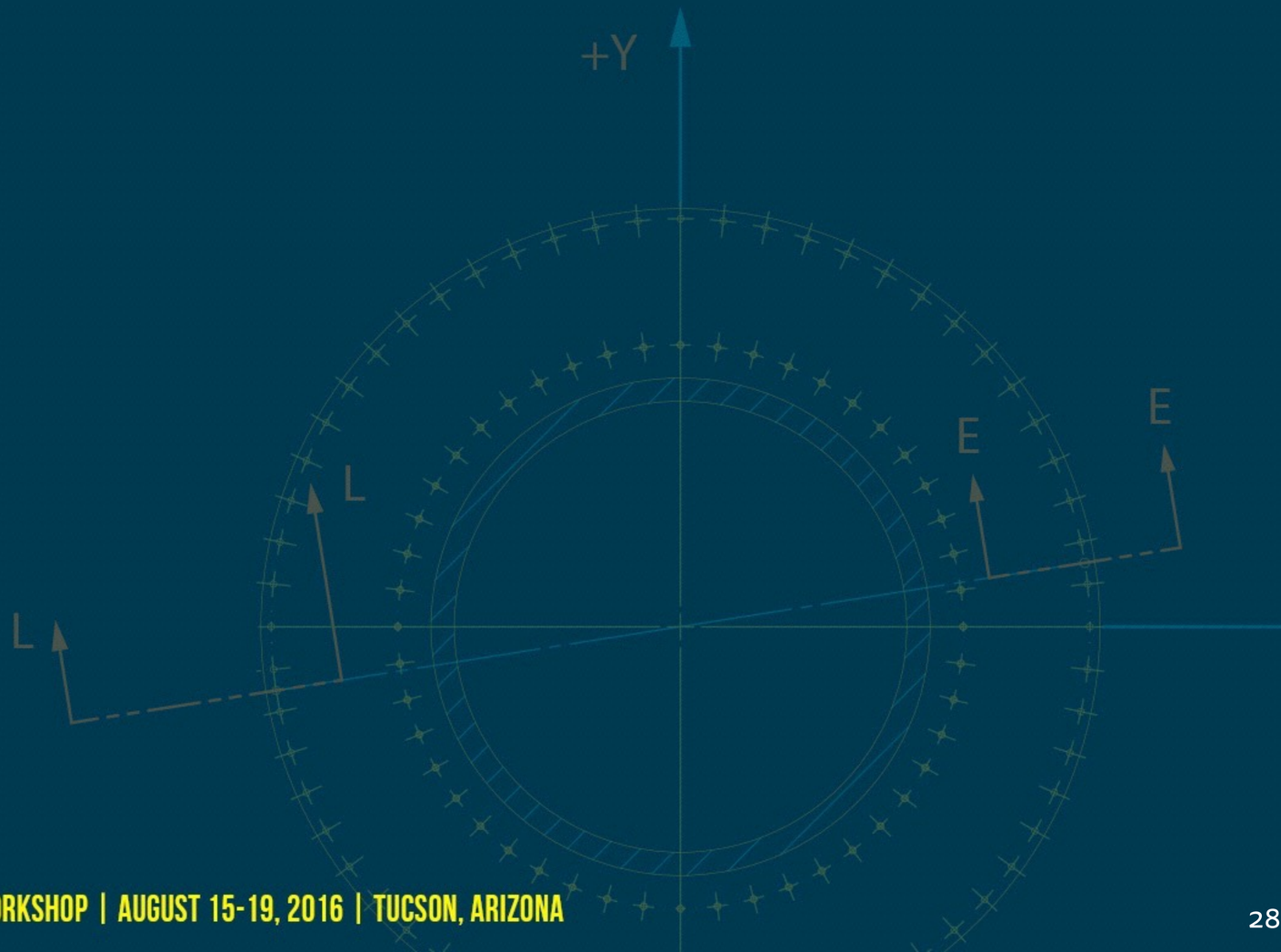
Classes

BST(data, row_index[, unique])	A basic binary search tree in pure Python, used as an engine for indexing.
Column	Define a data column for use in a Table object.
ColumnGroups(parent_column[, indices, keys])	
Conf	Configuration parameters for **astropy.table**.
FastBST	alias of **BST**
FastRBT	alias of **BST**
JSViewer([use_local_files, display_length])	Provides an interactive HTML export of a Table.
MaskedColumn	Define a masked data column for use in a Table object.
NdarrayMixin	Mixin column class to allow storage of arbitrary numpy ndarrays within a Table.
QTable([data, masked, names, dtype, meta, ...])	A class to represent tables of heterogeneous data.
Row(table, index)	A class to represent one row of a Table object.
SortedArray(data, row_index[, unique])	Implements a sorted array container using a list of numpy arrays.
Table([data, masked, names, dtype, meta, ...])	A class to represent tables of heterogeneous data.
TableColumns([cols])	OrderedDict subclass for a set of columns.
TableFormatter	
TableGroups(parent_table[, indices, keys])	
TableMergeError	

One sentence summary from function/class docstrings used in API tables.

Writing a clear one-sentence summary is critical.

Writing.



Writing is hard work

Writing is hard work. A clear sentence is no accident. Very few sentences come out right the first time, or even the third time. Remember this in moments of despair. If you find that writing is hard, it's because it is hard.

— William Zinsser, *On Writing Well*

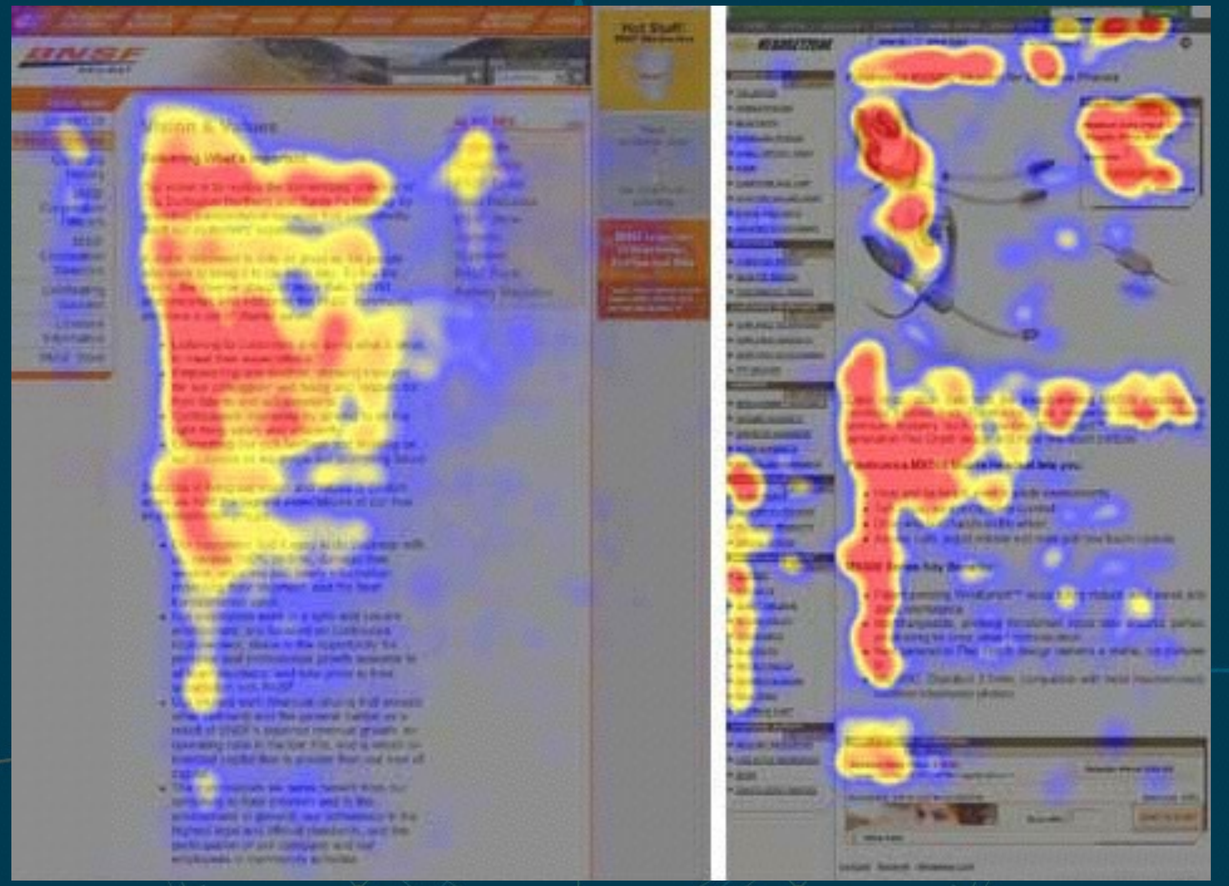
Design scannable docs

From: F-Shaped Pattern For Reading Web Content by Jakob Nielsen, http://ls.st/fzp.

"Users won't read your text thoroughly in a word-by-word manner.

"The first two paragraphs must state the most important information.

"Start subheads, paragraphs, and bullet points with information-carrying words.

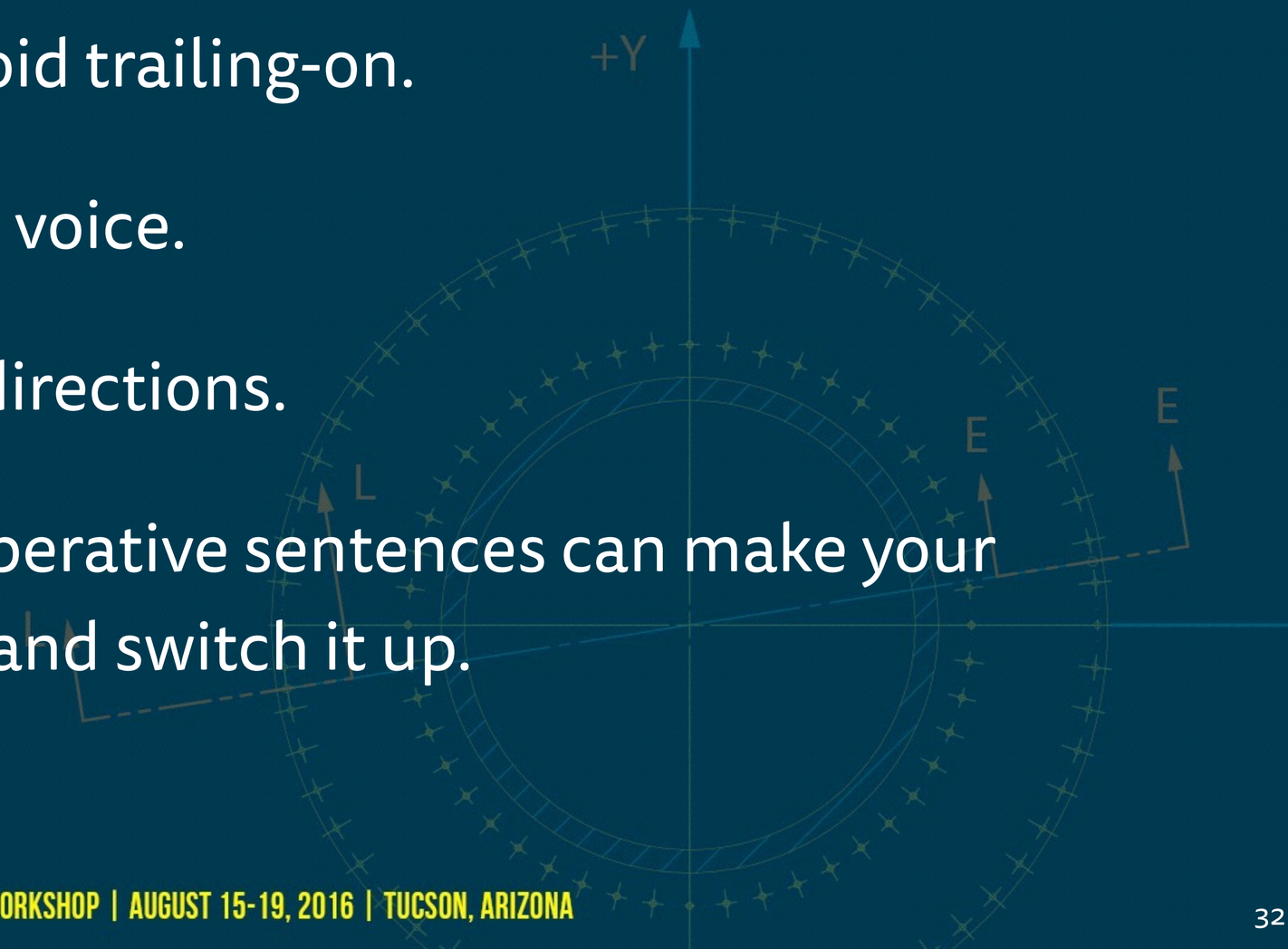


Design scannable docs

- Most people don't read the web sentence-by-sentence.
- Use lots of meaningful headers.
- 2–3 sentence paragraphs.
- First one or two words of a paragraph should convey meaning.
- Use lists and typographic elements for structure.

Craft sentences

- Write short sentences. Avoid trailing-on.
- Opt for active over passive voice.
- Use imperative for giving directions.
- But only using short or imperative sentences can make your writing cold. Use your ear and switch it up.

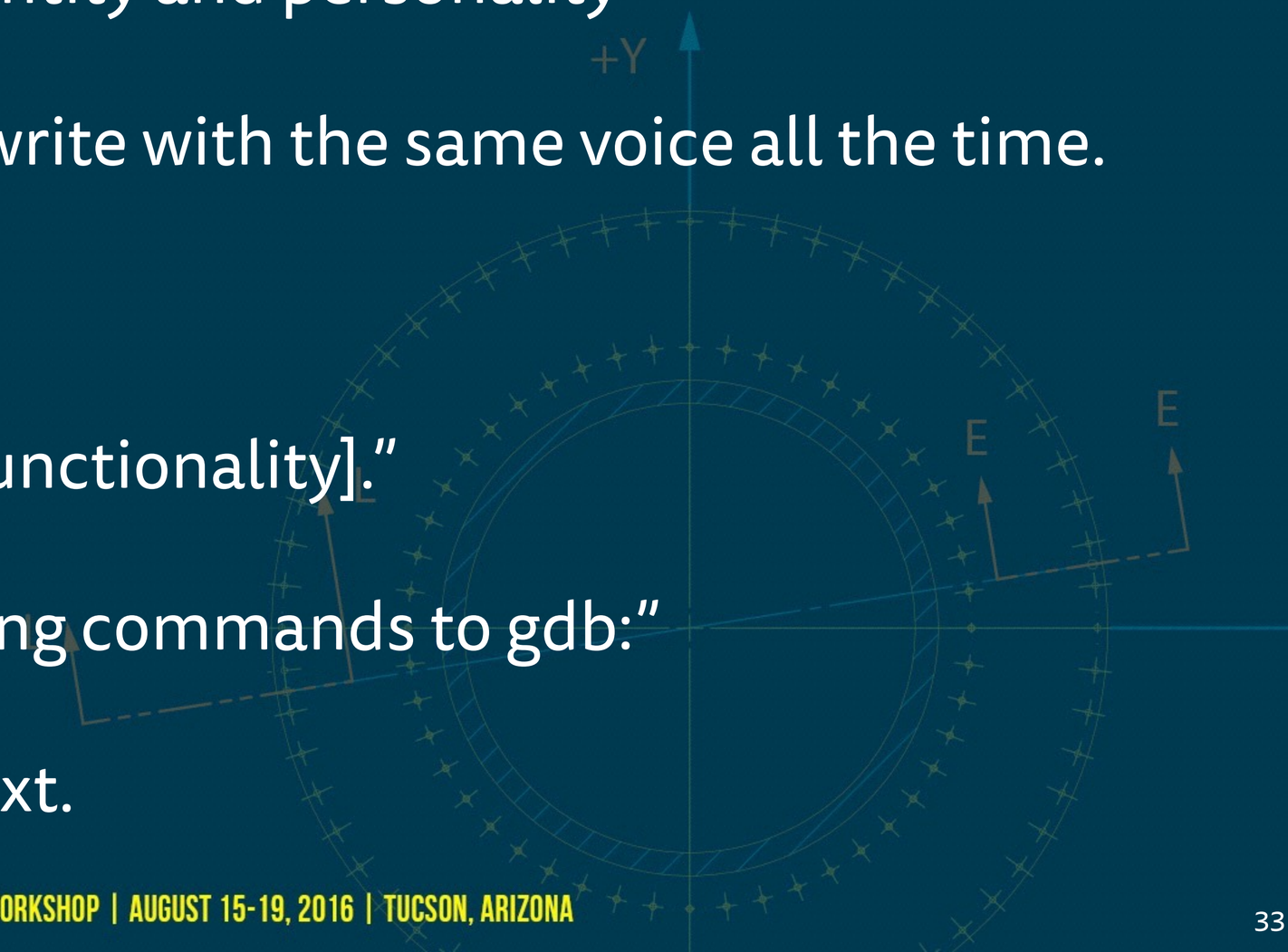


Voice & Tone

Voice is our organization's identity and personality

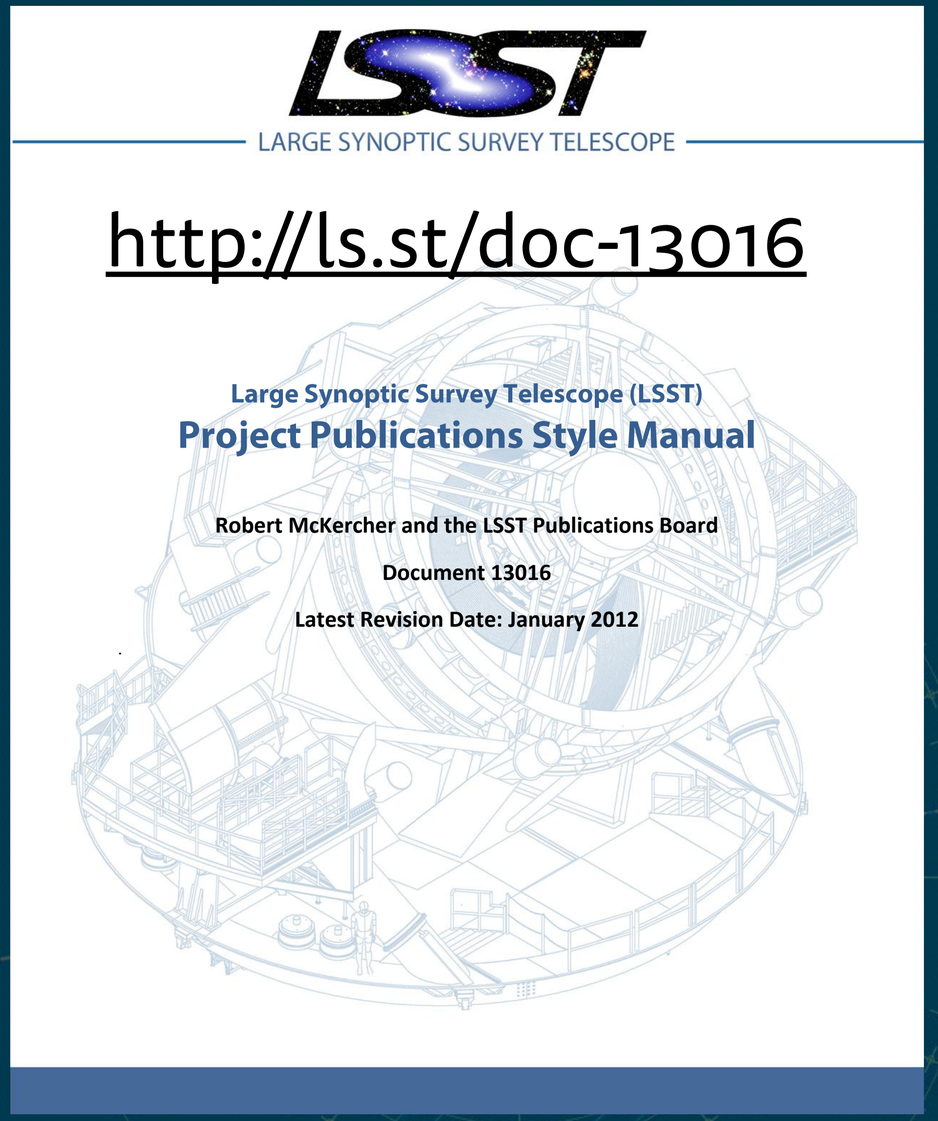
- As a group we [should] write with the same voice all the time.
- Is this our voice?
  - "[API] was taught to [functionality]."
  - "Then chant to following commands to gdb:"

Tone is how we write in context.

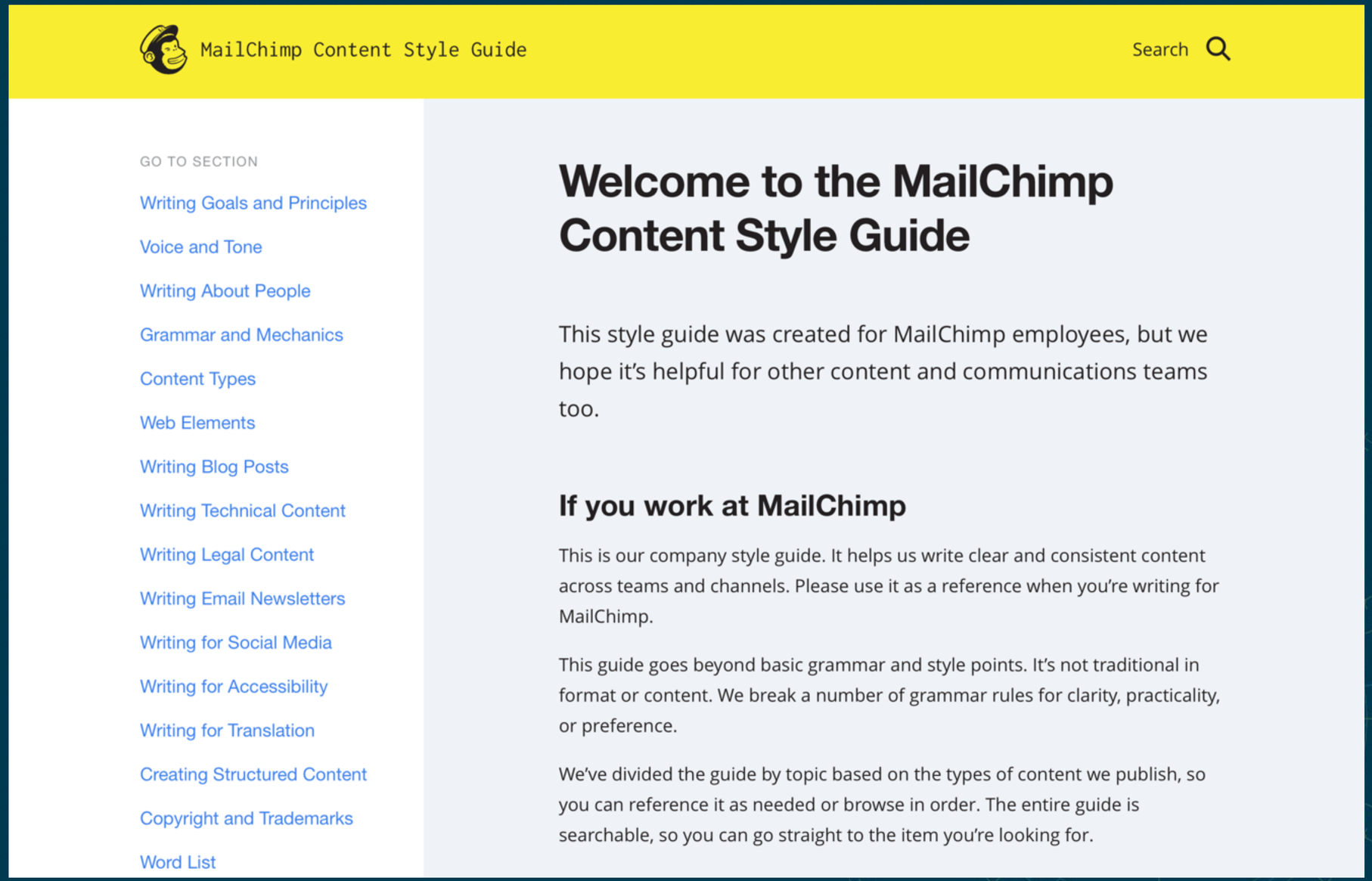


Voice

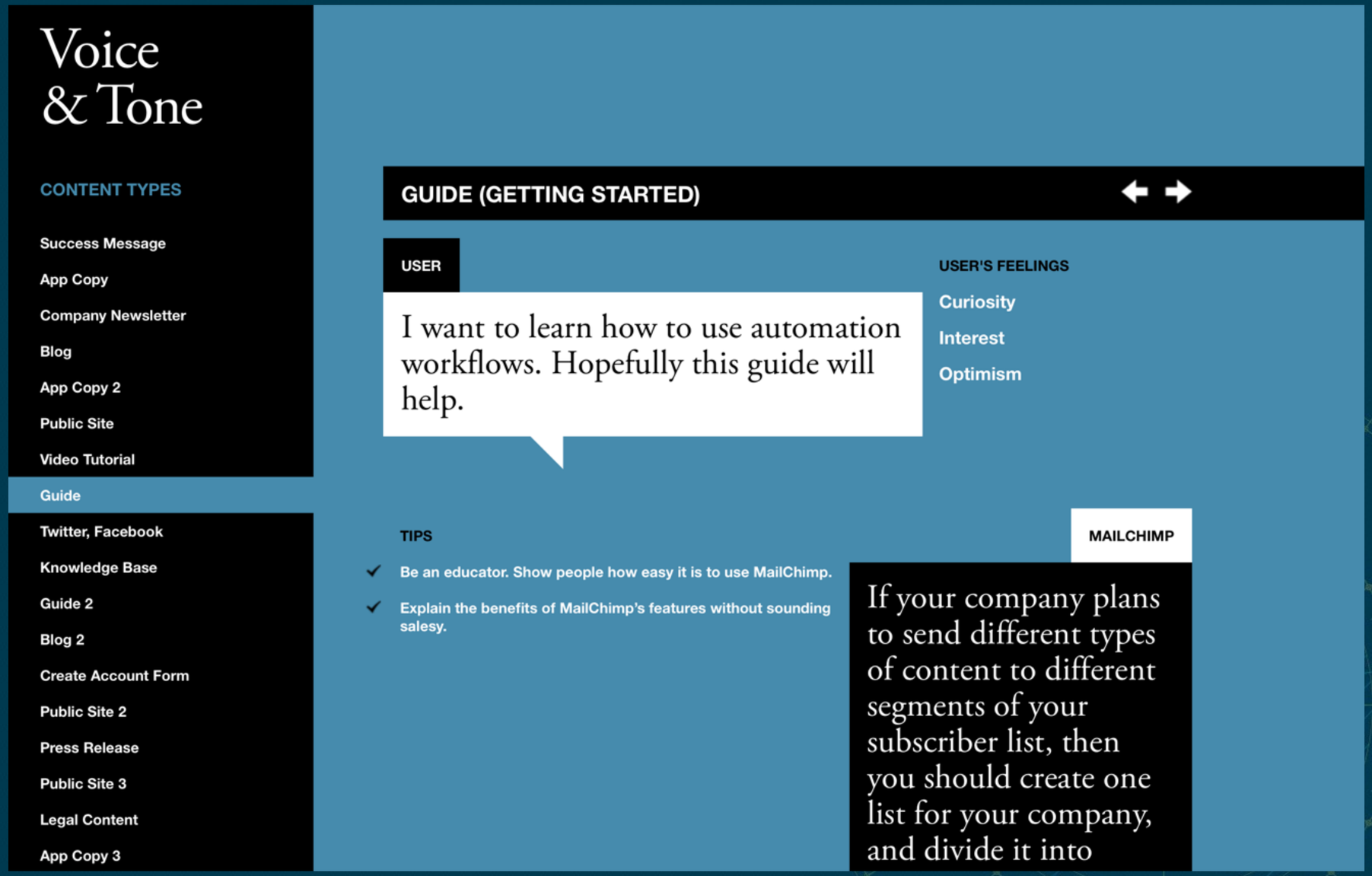
LSST DM could benefit from an addendum to the LSST Style Manual that gives vocabulary and phrasing guidance.



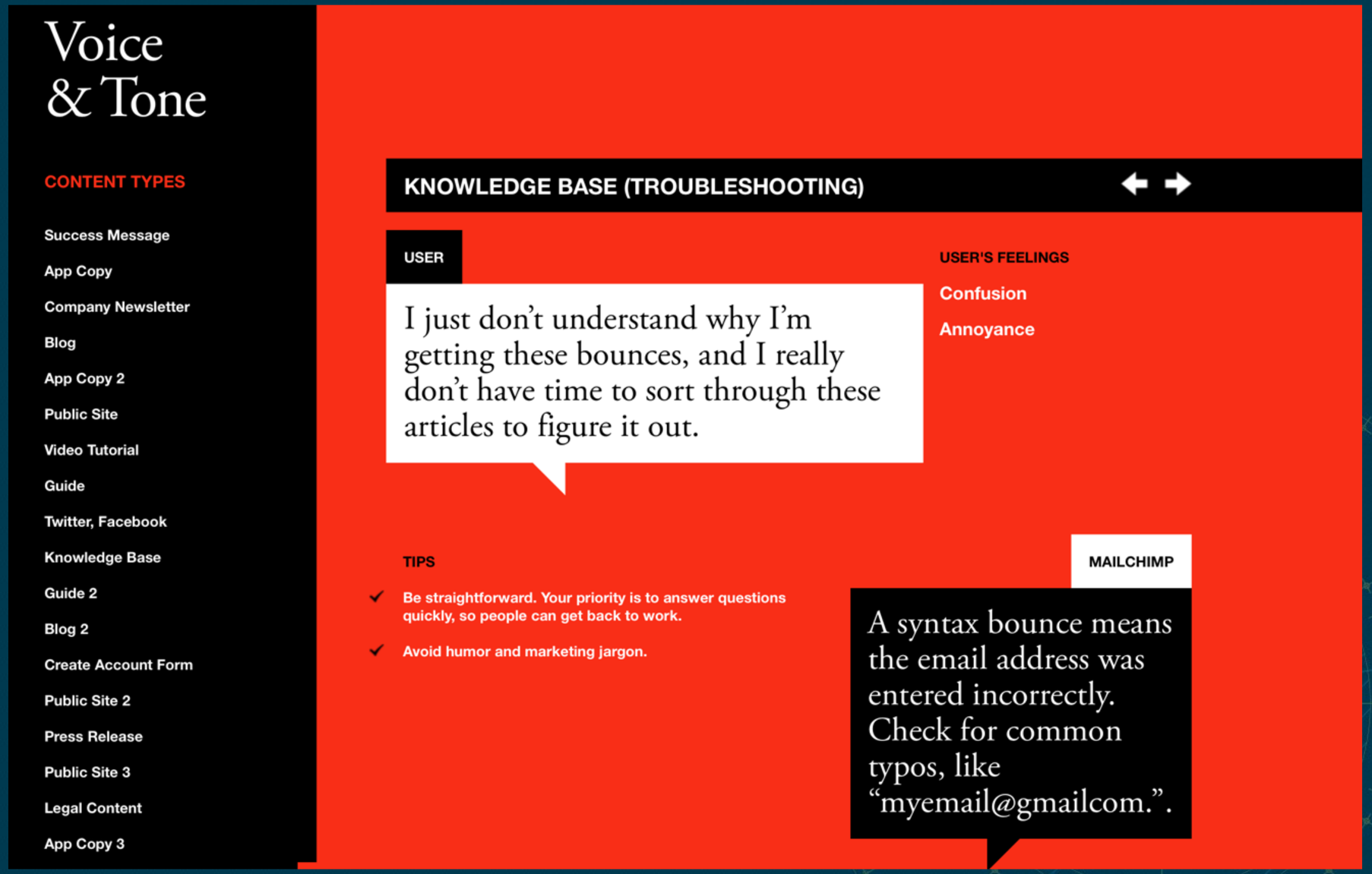
styleguide.mailchimp.com



voiceandtone.com

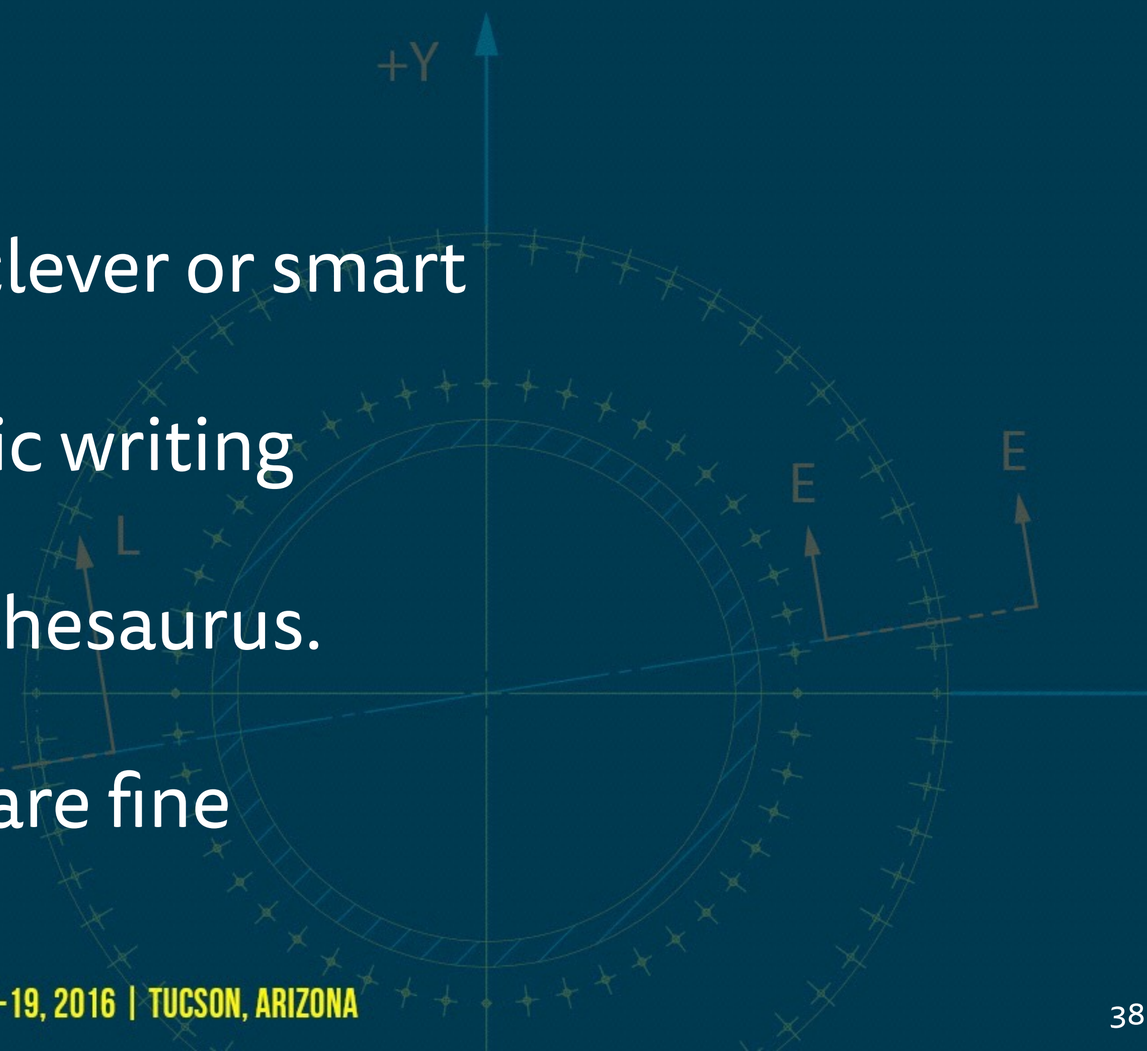


voiceandtone.com



Be conversational in tone and write simply

- Don’t go out of your way to sound clever or smart
- Documentation is not like academic writing
- Use 5 cent words. Throw out your thesaurus.
- Write like you speak; contractions are fine

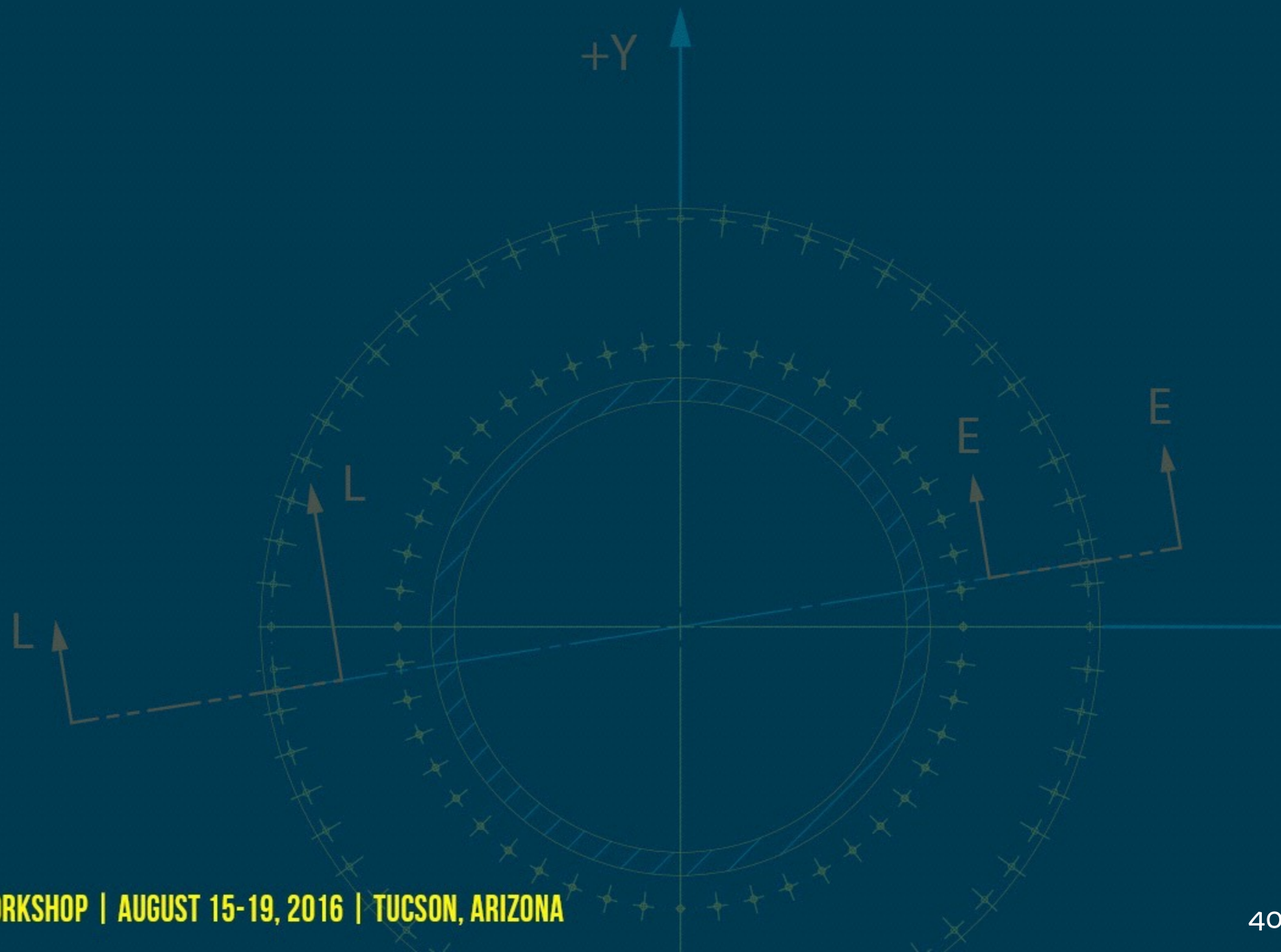


Be honest and considerate

- Avoid the words “easy,” “trivial,” and “just.”
- Your readers are intelligent; they just don’t have your domain knowledge yet
- Adapt your tone for error messages and other tricky situations

    Anticipate your reader’s emotions and context

Editing



Advice for Writers

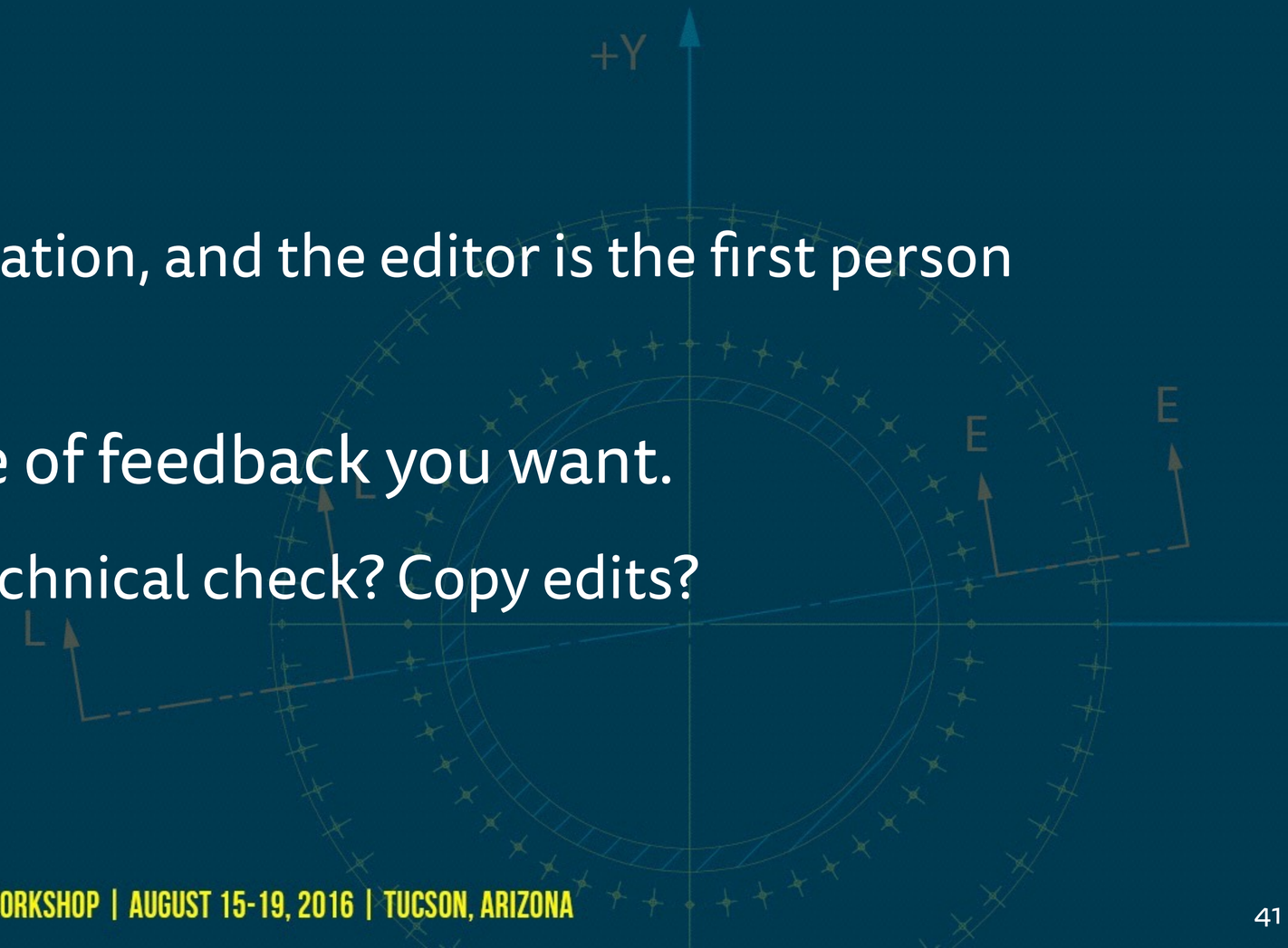
- Be open to feedback.

  Writing is about communication, and the editor is the first person you connect with.

- Be up-front about the type of feedback you want.

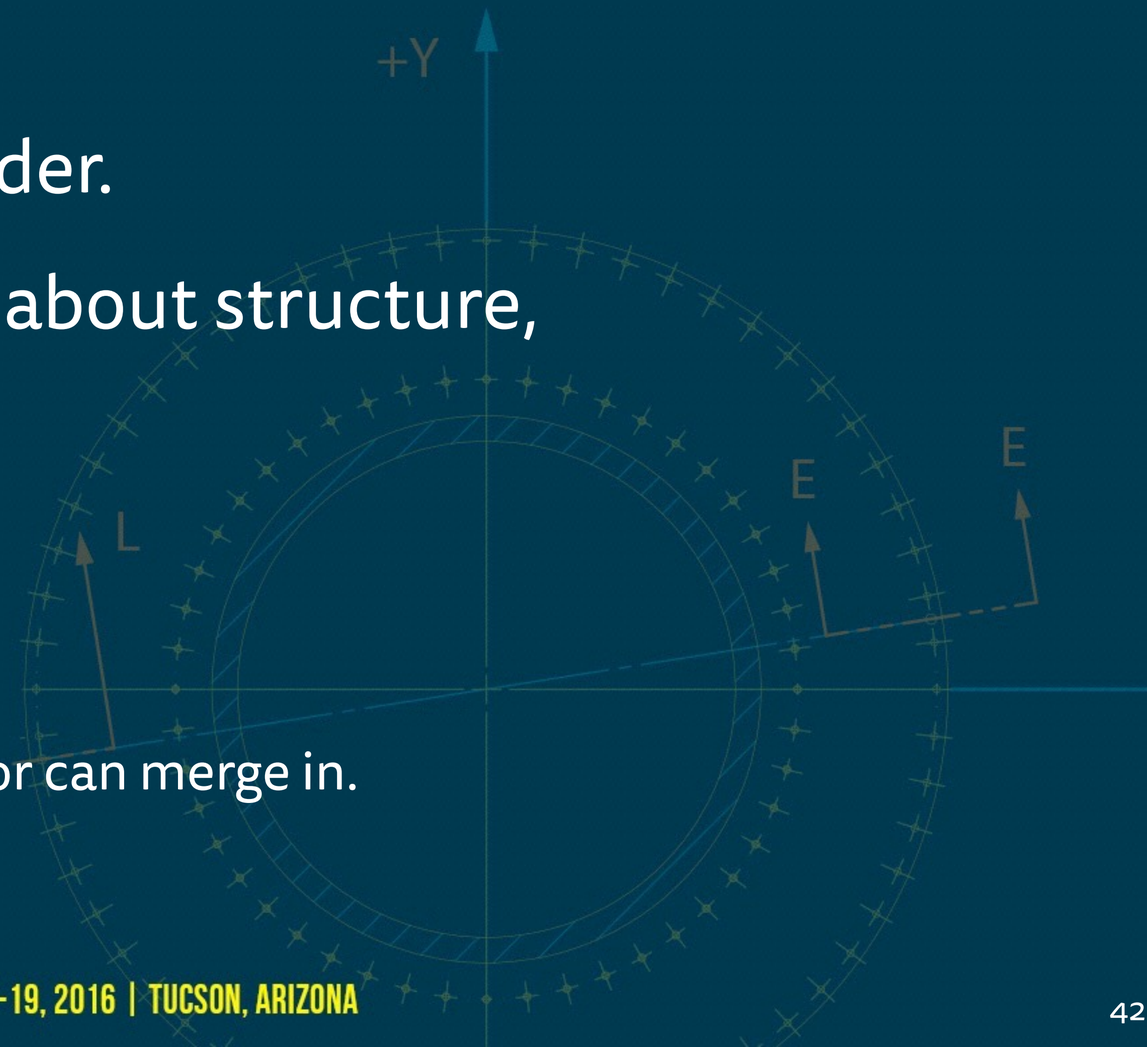
  Structural organization? Technical check? Copy edits?

- Editing is a conversation.



Advice for Editors

- Your job is to advocate for the reader.
- Use GitHub PR comments to talk about structure, flow, and content.
- Two options for copy editing:
  1. Edits in PR line comments.
  2. Commit edits on a new branch that the author can merge in.



Some resources

Books

- *Nicely Said* by Nicole Fenton and Kate Kiefer Lee. http://ls.st/6fl

Web

- Write the Docs conference videos. http://ls.st/4gu
- Jacob Kaplan Moss's series on documentation. http://ls.st/d6w
- *Teach, Don't Tell,* by Steve Losh. http://ls.st/2h2
- *Cooking up Effective Technical Writing* by Sally Jenkinson for 24ways. http://ls.st/owh
- Mailchimp's *Content Style Guide*. http://ls.st/9jo
- *Writing for the Web*. Dalhousie University. http://ls.st/i5d