Building Shiny Apps the BOAST Way

A How To and Official Style Guide

Neil Hatfield (njh5464@psu.edu) and Robert Carey (rpc5102@psu.edu)

19 November 2020

Contents

Welcome

One of the biggest advantages of the modern era is computing power. This provides with the ability to present statistical concepts in ways that allow students to explore and build their understandings in dynamic ways. A growing tool on this front is the advent of applications built in the `R` language and utilizing the `shiny` package by the RStudio company; these are called "Shiny Apps".

We are not the first group to build online applications to help students learn ideas, nor will we be the last. In fact, we're not even the first group to build Shiny apps for Statistics Education. Thus, there are many different approaches to how we go about designing and coding our Shiny apps. The primary goal of this document is to layout the approach that we use to 1) formalize the approach for ourselves, and 2) share our approach with others who are wanting to build their own Shiny apps for teaching.

This guide spells out the recommended approach for building and the required styling to should be used for all apps included in the Book of Apps for Statistics Teaching (BOAST). You will need to follow this Style Guide to ensure that any app you create meets (or exceeds) our standards is worth of inclusion in BOAST.

Effective Date: [to be listed here]
All apps created and/or modified after the Effective Date are required to adhere to this version of the Style Guide. Apps which have not been modified since before the Effective Date will be brought into compliance on rolling basis.

0.1 Organization

We've laid out this document into several parts, each with their own chapters.

- Part 1: Getting Started
  - Chapter 1—Explain Your Idea

  - Chapter 2—Using a Workflow

- Part 2: Getting Ready to Code

  - Chapter 3—Setting Up Your Computer

  - Chapter 4—Exploring the Boast Template
- Part 3: Style Guide–Coding
  - Chapter 5—BOAST Coding Style
  - Chapter 6—HTML and CSS
  - Chapter 7—Additional Coding Information
- Part 4: Style Guide–App Layout
  - Chapter 8—General Layout
  - Chapter 9—Common/Reoccurring Elements
- Part 5: Style Guide–Visual Style
  - Chapter 10—Color
  - Chapter 11—Static Images
  - Chapter 12—R Plots
- Part 6: Style Guide–Language
  - Chapter 13–Wording
  - Chapter 14–Documentation
- Part 7: Accessibility and Mobile Devices
  - Chapter 15–Accessibility
  - Chapter 16–Mobile Friendliness

Yes, this looks long but keep in mind the following:

> Building an app is easy. Building an app that someone else can use is a challenge.

> While building an app that supports a user in developing a procedural conception of an idea is easy, building an equitable app that supports anyone in developing productive meanings is more difficult.

Our mission with BOAST is to achieve both with our entire collection. Thus, writing downs all of the ins and outs of our process will take a significant amount of space. Whenever possible, we've attempted to provide you with examples to help you see our standards in action as well as code that you can use as templates. For each code block, you should be able to place your cursor over the block and see a copy icon in the upper-right hand corner. This will allow you to copy the code. We've also included links to additional resources so that you can learn more about key topics.

Keep in mind that this is a living document. As issues arise, we will update this guide to address them. Additionally, as we think of potential issues, we'll also update the guide to provide guidance before they occur. We also welcome any suggestions for improvements.

0.2 Why Our Approach?

While we could go on about the many reasons for our approach, we will focus on the critical ones:

0.2.1 We're a Team of Developers

From the start, the Book of Apps for Statistics Teaching (BOAST) has been the joint effort of faculty members and students. Having a clear articulation of how to make apps within our environment is critically to ensure that we have cohesiveness between the apps and through the years as the team changes. Our approach is meant to maximize the team's ability to adapt, maintain, and grow BOAST throughout the years.

0.2.2 Research Driven

We place a premium on drawing from what researchers have found about student thinking on various topics. And for those topics where the literature is thin or non-existent, we can fall back on the wealth of experience and expertise our faculty have. When we design apps, we focus on what the learning objectives are and work on how we might support students in developing meanings that are consistent with our targets.

0.2.3 Accesiblity Minded

One of the key ways in which our approach differs from a vast majority of others is the level of accessibility thinking we go through. Shiny apps are **not** very accessible by default and are extremely easy to make even more inaccessible.

For example, consider the following entries in the 2020 Shiny app competition, as of 10/15/2020:

- Blog Explorer by Stefan Schliebs: This Grand Prize winner has 11 errors, 68 contrast errors, and 14 alerts.
- Life of Pi: A Monte Carlo Simulation by Zauad Shahereer Abeer: This Honorable mention has 17 errors, 13 contrast errors, and 9 alerts.
- OTS Beta Dashboard by Mauricio Vargas: This Honorable mention generates 14 errors, 0 contrast errors, and 5 alerts.
- Probability of Normal Distribution by Aep Hidayatuloh: Produces 5 errors, 108 contrast errors, and 27 alerts.
- Didactic Modeling Process: Linear Regression by Daniel Rivera: while not part of the Shiny contest, this is a currently featured app in RStudio's Shiny User Showcase, which has 57 errors, 26 contrast errors, and 184 alerts.

We used WebAIM's Web Accessibility Evaluation Tool (WAVE) to get these measurements. However, we must point out that the above numbers are *under*-estimates. The nature of a Shiny app interferes with WAVE's scanning tool (for example, non-displayed app pages aren't checked) as well as the fact that rendered R plots can only be checked for the presence of alt text/aria labels but not contrast errors, small font sizes, etc.

Now, this isn't to say that our problem free. However, our process ensures that the number of errors that exist are in the single digits. When we publish an app, most of the remaining errors are ones we inherent from the `shiny` package and we're striving to address.

0.2.4 Avoid the "Shiny" Problem

In Education, and particularly in the American system, we wrestle with the "silver bullet problem". This problem is the belief that some reform effort will function as a fix or cure-all for the issues facing the education system. These reforms never bare out.

In design, people can get caught up with new (at least to them) tools such as a glitzy new system to do something. They will then start incorporating it everywhere they can. However, they often don't stop to think about whether or not this glittery new toy is appropriate or sound design. This is especially true for novice designers.

When dealing with Shiny apps for teaching, we face both problems simultaneously, which we call "The Shiny Problem". We work hard to ensure that

1. Our apps aren't going to be viewed as silver bullets but as useful resources that can supplement instruction
2. Just because `R` can do something doesn't automatically mean that we're going to include it; there must be a sound educational and pedagogical reason for including it.
3. Just because there's a fancy new package that's been released doesn't mean we'll blindly adopt it and use it wherever; there must be a sound educational and pedagogical reason
4. Just because someone else made an app that did such-and-such doesn't we mean we're going to; there must be a sound educational and pedagogical reason.

A key exception to the above deals with coding and app performance, but these are secondary to sound education and pedagogical reasons. We strive for our Shiny apps to not become *shiny*.

Part I

Getting Started

Chapter 1

Making an App

Sooner or later, you'll reach the point where it is time to make your own Shiny App for BOAST. Approaching your App's construction in a systematic way will help you tremendously. The following suggested workflow is an abbreviated version of the one in Section 2:

1. Read the Style Guide
2. Identify a topic
3. Sketch out your plans
4. Create a new repository on GitHub
5. Begin writing the code
6. Edit and locally Test your code
7. Push your code to GitHub/Create a dev branch
8. Push your edits
9. Larger scale testing
10. Pull Request
11. Additional tweaks

This workflow (both in the more expanded state and in the abbreviated one) leaves out a lot of the going-ons and decision making that goes into making an app. Our goal with this section is to walk you through this process as Neil makes an app from scratch.

1.1 Step 1: Read the Style Guide

This is step is critical to making an app that is easy to debug (vital when programming) and is in-alignment with the Standards that we have set for BOAST. Please take your time going through the Style Guide and keep an eye on it for updates. You can access the published Bookdown version by clicking on the address link located at the top of the Guide's GitHub Repository.

As you build your App, we recommend periodically checking the Style Guide to 1) ensure that you're still adhering to it, 2) to stay up-to-date with the Guide, and 3) to see (and then use) any useful code chunks that are listed. Even as the authors of this Guide, we're constantly looking things up. Thus, there is no shame in constantly referring to the Style Guide; after all, it is meant to be a Go-To Reference.

1.2 Step 2: Identify a Topic

This is one of the more challenging tasks for anyone: deciding what to make an app about. The notion of "picking a topic" is a bit misleading as not only should you pick a topic, but you also need to pick the purpose of the app, and the goals. While all three go hand-in-hand with each other, the easiest decision of the three is purpose.

1.2.1 Your App's Purpose

When we look across BOAST, we can categorize the apps into two major classes: apps whose purpose is to help students **learn** an idea/concept and apps whose purpose is to help students **review** an idea/concept. The NHST Caveats App stands as a good example of an app whose purpose is to help students learn something. On the other hand, the [Null] Hypothesis Testing Game (Tic-tac-toe) and the Matching Distributions are examples where the student is reviewing their understandings. There are some apps such as ANCOVA that are dual purpose, having both learning elements and reviewing elements.

Generally speaking, apps whose purpose is **learning** are going have Explorations and possibly Challenges for Activity Tabs. Apps whose purpose is **reviewing** will have just Games for Activity Tab(s).

By identifying early on the purpose of your App, you'll be able to better plan your app.

1.2.2 The Topic

Here is where you'll decide what statistical idea/concept your App is going to be centered around. While you should be as specific as possible here, **apps for learning** need to have much more specificity than **apps for reviewing**. Saying that you want your App "to deal with probability" is not going to be as useful as saying that you want your App "to help students recognize when probability is and is not appropriate for different contexts".

Don't feel like you have to details to the nanoscale. As you continue developing and talking with people, you'll start adding in more details.

You can identify potential topics through any of the following:

- Think of a concept in a Stat class that you (or your peers) struggled with (or still do)
- Think of a concept where you thought to yourself “I wish there was a way to visualize this”
- Think of a concept where you went “How does this work?”
- Did you see/hear about a new study and thought to yourself “what’s going on?”
- Did a media report make you do a double take?
- Ever think to yourself “I wish I had more practice with [topic].”
- And more

Again, this list is just a starting place to identify potential topics. Draw inspiration from your experiences, your curiosity. Bounce ideas off of each other and the faculty.

Word of Caution #1: While it is okay to get ideas from other websites, please keep track of this. We need to give credit where credit is due. If you come across an app (whether in Shiny or some other program) somewhere and you think that it would be good to translate that into Shiny app for BOAST, we need to know.

Word of Caution #2: An unproductive place to begin is to say “Hey, I want to make a memory game” or “I saw this really cool animation and I want to make an app that does that.” Both of these (especially the second) place the educational aspect of the app as a secondary focus. **This is not consistent with BOAST.** All apps should place educational purposes and goals before anything else. Forcing a topic to make use of certain tool because that tool would be “cool” or “fun” does not further our goals. We should strive for synergy between topics and the tools within Shiny.

1.2.3 Goals

Now, we’re sure that some people have already thought to themselves “hey, you’ve already had us identify purpose so isn’t this the same?” To which we reply “No.” We’ve used **purpose** to refer an overarching aim of your App. Here, **goal** refers to a much more contextualized aspect. While these goals will be less important for reviewing apps, you will need to identify specific learning goals for learning apps.

For this step, you’re going to need to work with the faculty to identify the learning goals for your app. If you’re drawing from a past experience in a course, look to see if there were learning objectives listed. Those can be extremely useful starting places.

1.2.4 Example

The following is a somewhat stream-of-consciousness narrative of how Neil tackled this step. We’ll place flags in parentheses.

I would like to make an app that would help introductory students (*identifies an intended audience*) wrestling with some of the concepts of descriptive/incisive statistics (*Purpose: learning*). Some ideas that I have are:

- Conceptualizing the two distinct uses of the *sample arithmetic mean*–mitigation of measurement errors of a single object vs. measuring how well a group of objects/beings performs
- Conceptualizing statistics as functions that measure attributes of collections
- Conceptualizing the relationship between the values of the *sample median* and *sample arithmetic mean* is not a competition (i.e., one is not "better than" the other)
- Avoid falling into the skewness trap for the ordering of the values of the *sample median* and *sample arithmetic mean*.

This last option feels like a fairly straightforward (and simple) app to create. Additionally, there's an article by von Hippel on this issue I can reference. (*Topic: relationship between the values of* sample median *and* sample arithmetic mean *in the presence of skewness*)

After using the app, a student should recognize that while for a many data collections that have skewness, the value of the *sample arithmetic mean* will be either greater (for positive skew) or less (for negative skew) than the value of the *sample median*, this is not an absolute rule. (*Learning Goal*)

1.3 Step 3: Sketch out plans

Something to keep in mind with this workflow is that it is not strictly linear. That is, just because you have moved into Step 3, doesn't mean that you can't go back to Step 2 and make revisions. Additionally, as you get into Steps 5-8, you might find yourself coming back to Step 3 (and possibly Step 2) and making changes. This said, sketching out your plans will help you as you move forward with developing.

There are several things that you'll want to be sure that you include in your sketch:

- A Suggested Title
  - This might not be what you settle on, but you'll need to have this firmed up by the time you get to Step 4.
  - You'll want to have both a [Long/Formal] Name and a shortened name ready
- Goal(s) of the App
  - Carry this forward from Step 2 as a tangible reminder
  - If you have several goals, you might consider denoting which tabs go with which goals
- What will the user be doing

  - Thinking about what you want the user to do
  - Identify anything the user might click on, drag, or otherwise manipulate.
  - If there are going to be options, identify what those options should be.
  - Will there be different levels?
  - Will there be a Challenge tab?
- What will the user experience
  - Think about the general layout (explore, game, etc.)
  - What outputs should the user be examining
- What are the relationships between elements
  - Think about how each input impacts the outputs
  - If you want the user to have a certain experience, does the coding allow for that experience to happen?
- Other Elements
  - Are you going to reference any external images?
  - Are you going to import any data files? If so, which ones? Do you have them?
  - Is there an activity packet?

The format of your sketches is up to you.

1.3.1 Example

As I start thinking about what this app might look like, I'm going to use a sheet of paper in quarters to organize some thoughts. Figure 1.1 shows the entire sheet of paper. The quadrants move left to right, top to bottom.

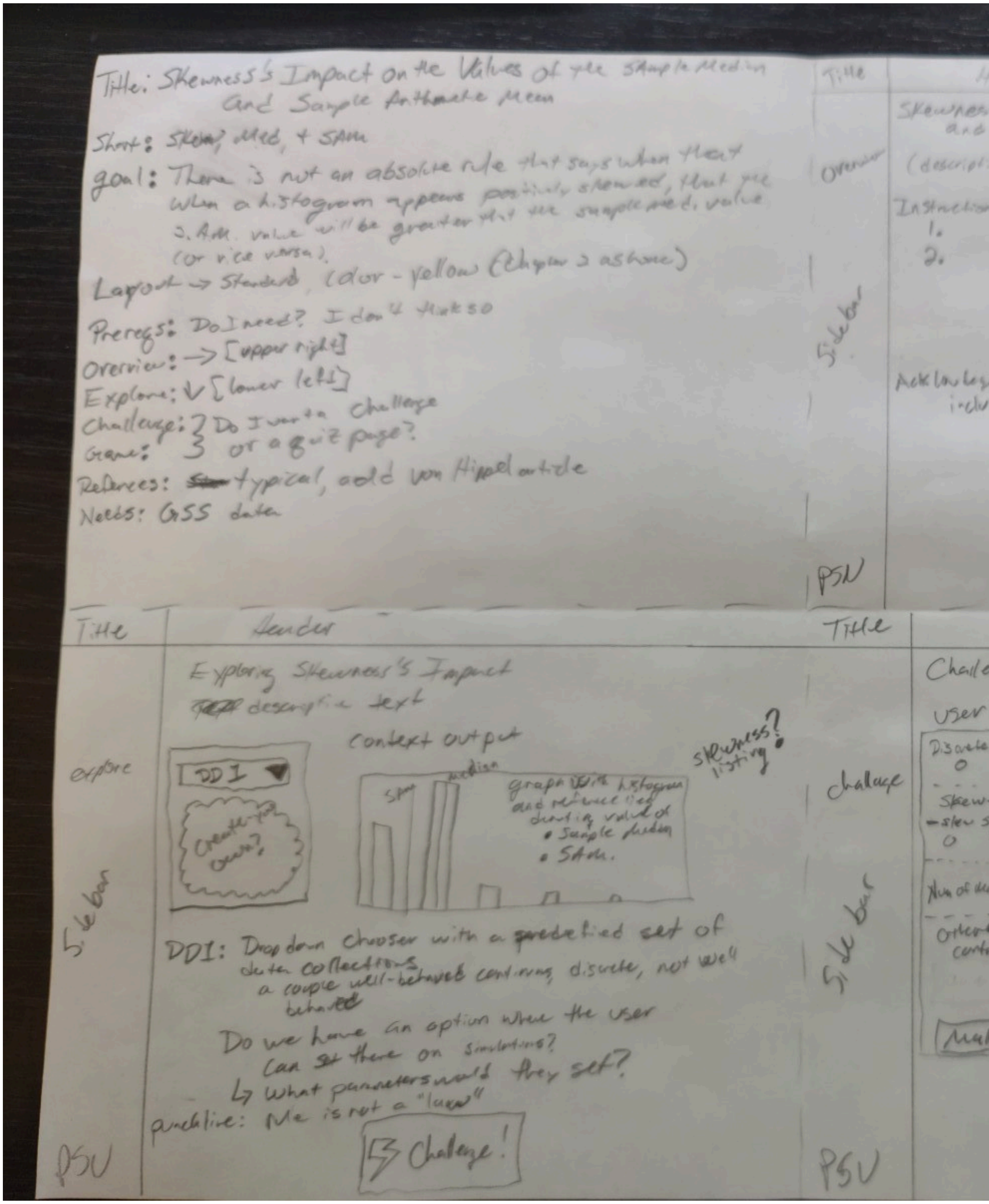
In the first quadrant (Figure 1.2), I've listed a title ("Skewness's Impact on the Values of the *Sample Median* and *Sample Arithmetic Mean*"), a possible short title ("Skew, Med, SAM") as well as a restatement of my goal. I've then listed general aspects about the layout of my app, including what skin/theme color (yellow) I would need to use.

I've additionally identified that I need to include the von Hippel article in my Reference Tab as well as locating a version of the General Social Survey (GSS).

The second quadrant of my paper is the Overview Tab sketch (Figure 1.3. I don't have much information here.

The next quadrant (Figure 1.4) is where I started detailing the first Activity Tab of my app: the first Exploration. I'm imagining that in this exploration that the user would look through a variety of cases where the "rule" does and does not work. What I want the user to walk away from this tab is that just because a histogram looks positively skewed does not imply that the value of the *sample arithmetic mean* is greater than the value of the *sample median*.

I'm imagining a drop-down menu that would allow the user to move through a variety of data sets. Ideally, these data sets would be real data. The General



Title: Skewness's Impact on the Values of the Sample Median and Sample Arithmetic Mean

Short: Skew, Med, + SAM

goal: There is not an absolute rule that says when that when a histogram appears positively skewed, that the S.A.M. value will be greater that the sample med. value (or vice versa).

Layout → Standard, Color - yellow (Chapter 2 as home)

Prereqs: Do I need? I don't think so

Overview: → [upper right]

Explore: ↓ [lower left]

Challenge: } Do I want a challenge

Game: } or a quiz page?

References: ~~Std~~ typical, add von Hippel article

Needs: GSS data

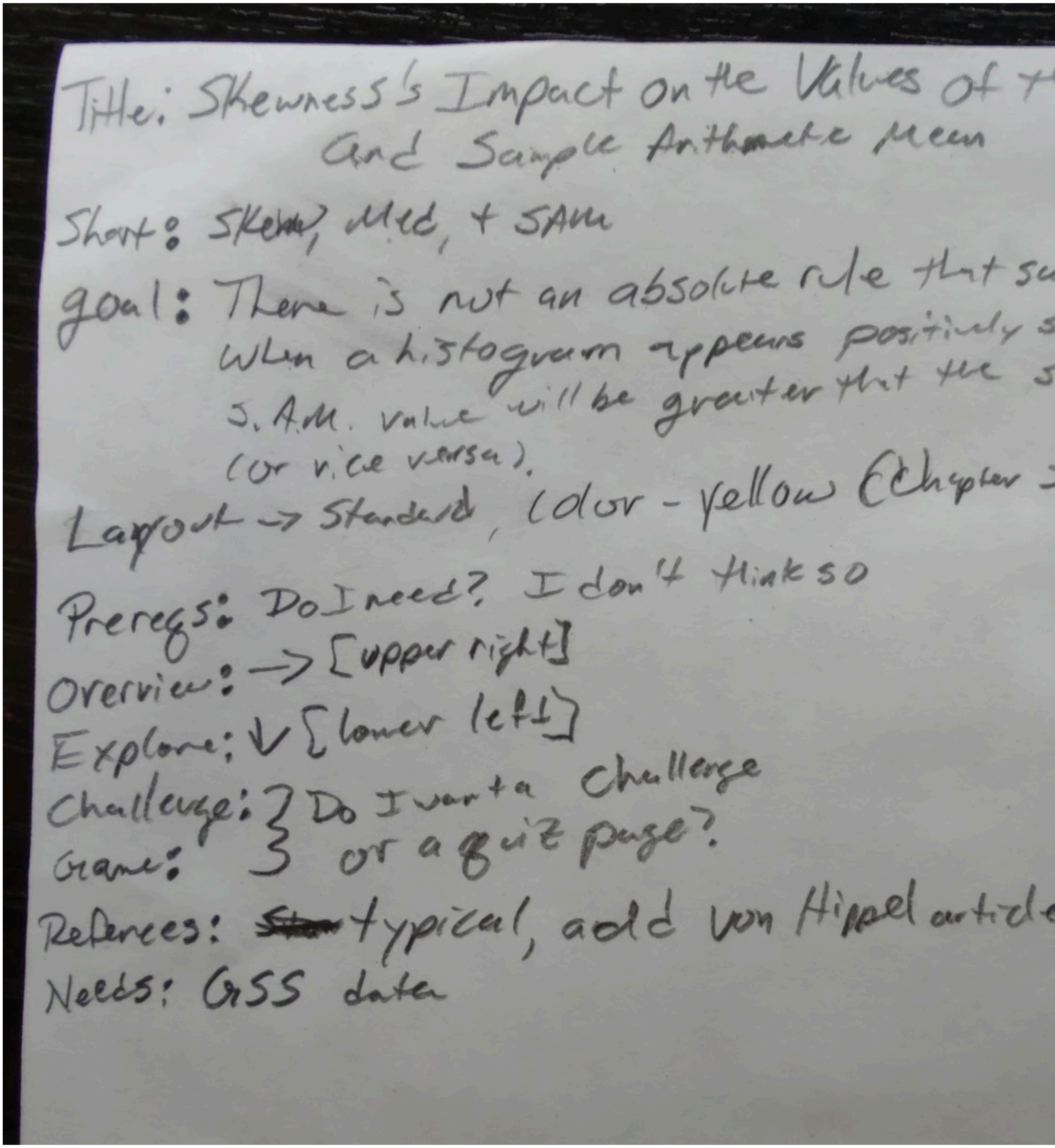
graph with histogram and reference lines denoting value of
- Sample Median
- SAM.

DDI: Drop down chooser with a predefied set of data collections
a couple well-behaved continuous, discrete, not well behaved

Do we have an option where the user can set there own simulations?
↳ What parameters would they set?

punchline: Me is not a "law"

Figure 1.1: Picture of My Plan



Title: Skewness's Impact on the Values of t
And Sample Arithmetic Mean

Short: Skew, Med, + SAM

goal: There is not an absolute rule that sa
When a histogram appears positively s
S.A.M. value will be greater that the s
(or vice versa).

Layout -> Standard, color - yellow (chapter

Prereqs: Do I need? I don't think so

Overview: -> [upper right]

Explore: ↓ [lower left]

Challenge: } Do I want a Challenge
Game: } or a quiz page?

References: ~~Stan~~ typical, add von Hippel article

Needs: GSS data

Figure 1.2: First Quadrant: General Information

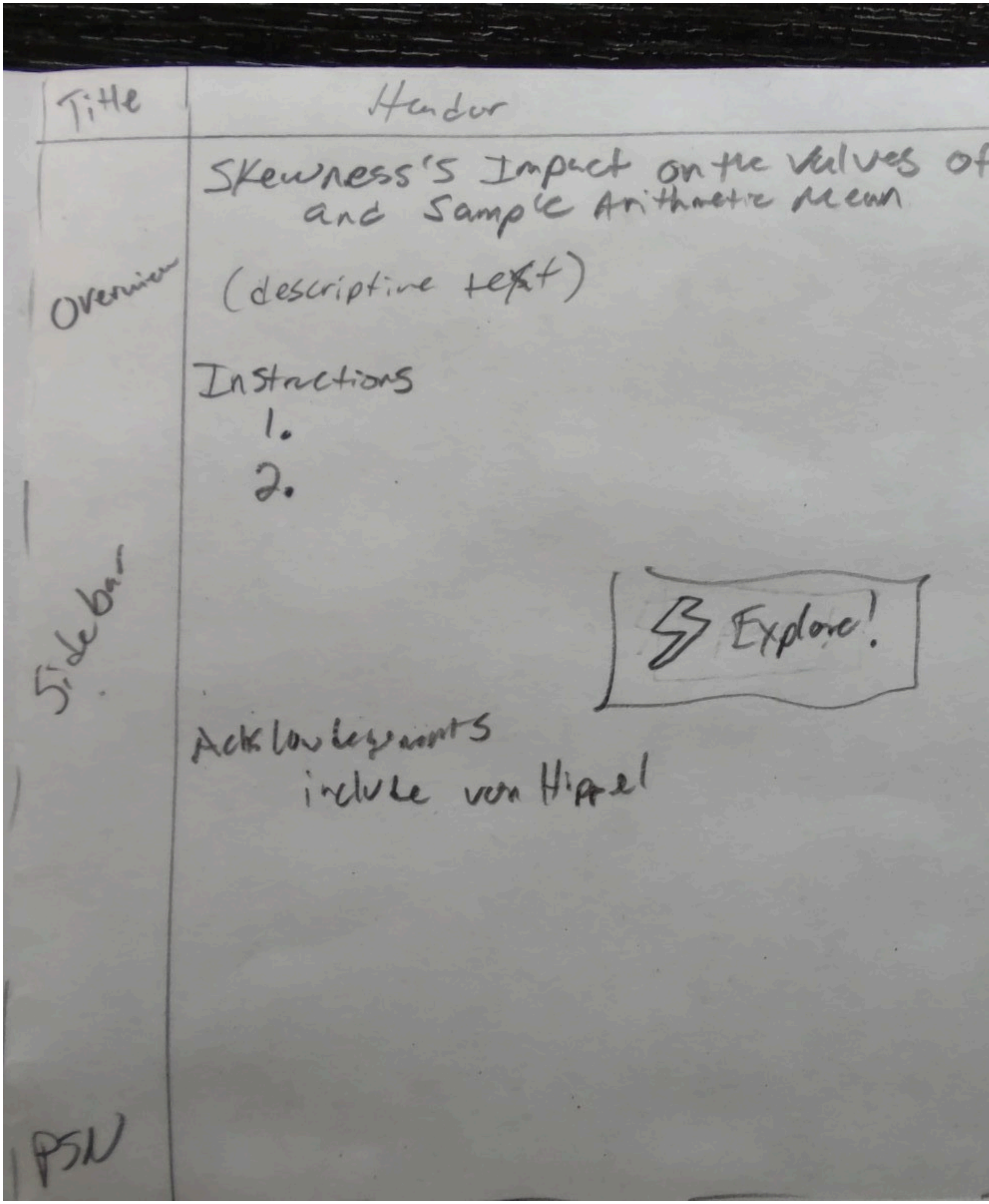
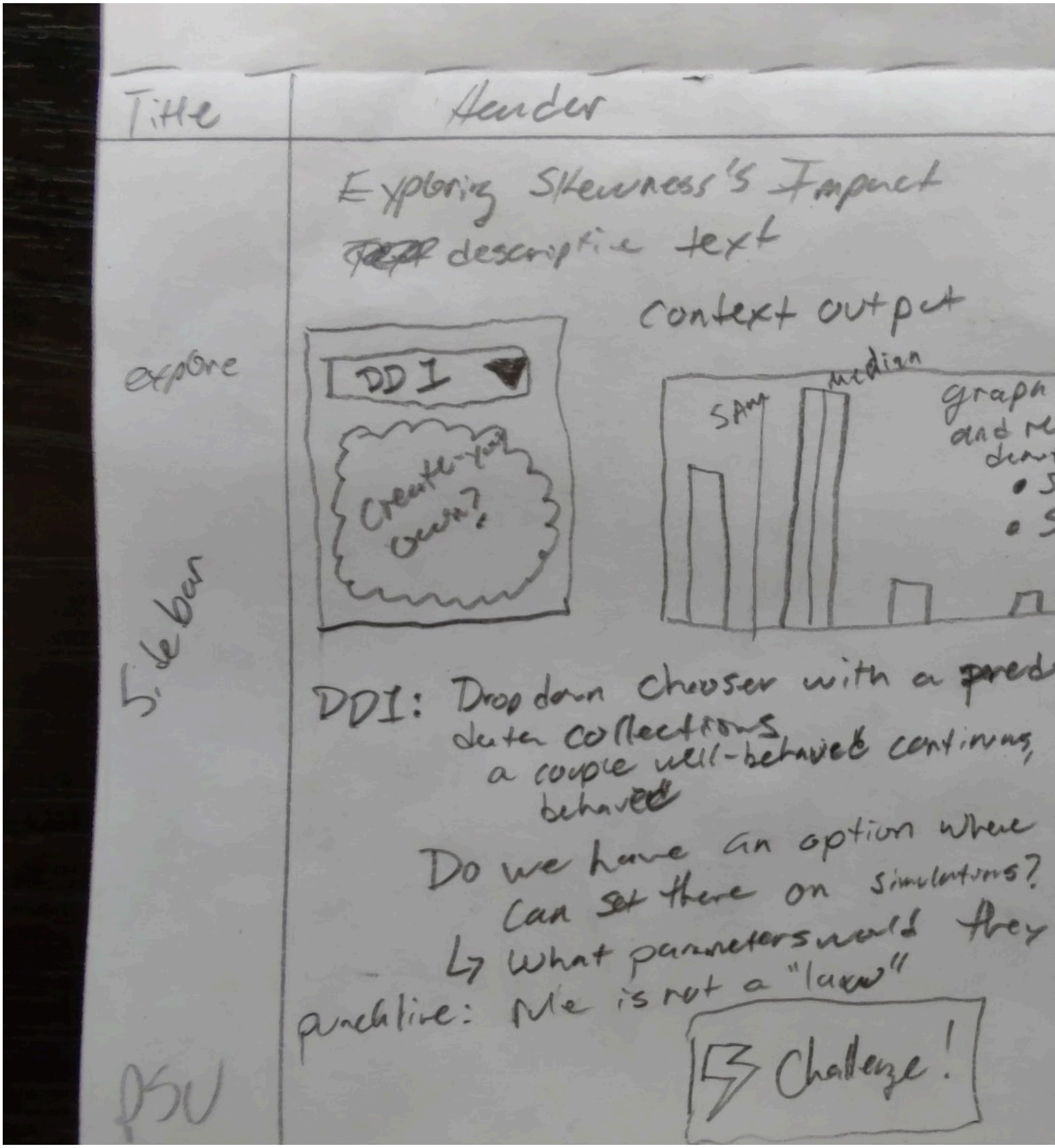


Figure 1.3: Second Quadrant: Overview Tab



DDI: Drop down chooser with a ~~pred~~
data collections
a couple well-behaved continuous,
behaved

Do we have an option where
can set there on simulations?
↳ What parameters would they

punchline: Me is not a "law"

Figure 1.4: Third Quadrant: Exploration Tab

Social Survey, has a variety of attributes that we could use, include several that demonstrate what I want. For example, the number of adults (18+ y.o.) living in a household is positively skewed but the value of the *sample median* is larger than than the value of the *sample arithmetic mean*. Which each choice of the user makes, the histogram on the right will update to show the appropriate data, plus reference lines for the values of the two statistics. I might want to add a numerical display as well. I also want there to be some context helping the user to understand each choice.

I've left as an open question whether to enable a user to import/create their own data set on this tab. I'm currently leaning towards not allowing this. Additionally, I've marked something for a punchline but this will need more thought.

The fourth and final quadrant (Figure 1.5) is an idea I'm still kicking around. This would depend on the specific audience and course goals for which users would go here. My idea is to extend the goal for the app to allow students to construct an understanding for what conditions might lead to $Sample\ Median\left(\mathcal{X}\right) < SAM\left(\mathcal{X}\right)$, $Sample\ Median\left(\mathcal{X}\right) = SAM\left(\mathcal{X}\right)$, and $Sample\ Median\left(\mathcal{X}\right) > SAM\left(\mathcal{X}\right)$.

At this moment, I'm leaning towards marking the Challenge Tab as being a future improvement/addition rather than being part of the initial version of the app.

1.3.2 Share Your Plan

Once you have a sketch, you'll want to share this plan with others to get ideas for suggestions, modifications, etc. Different people might be able to point you to code snippets you can reuse from other apps. More importantly, they might be able to help you think about things that you glossed over/missed and/or thought about but didn't articulate. By sharing these plans, we'll make better apps.

This is also the point in time in which the faculty will say whether you can go ahead and move to Step 4.

1.4 Step 4: Create Repository

Here you'll want to reference Section 2.2.1 on GitHub Repo Naming. Most app [full] titles don't make good Repo Names. This is where the potential short title comes into play. You'll need to make sure that there isn't already a repository that has that name. When you get a name figured out for the repository, you can go ahead and create the repo.

Make sure to set Git Ignore to `R`.

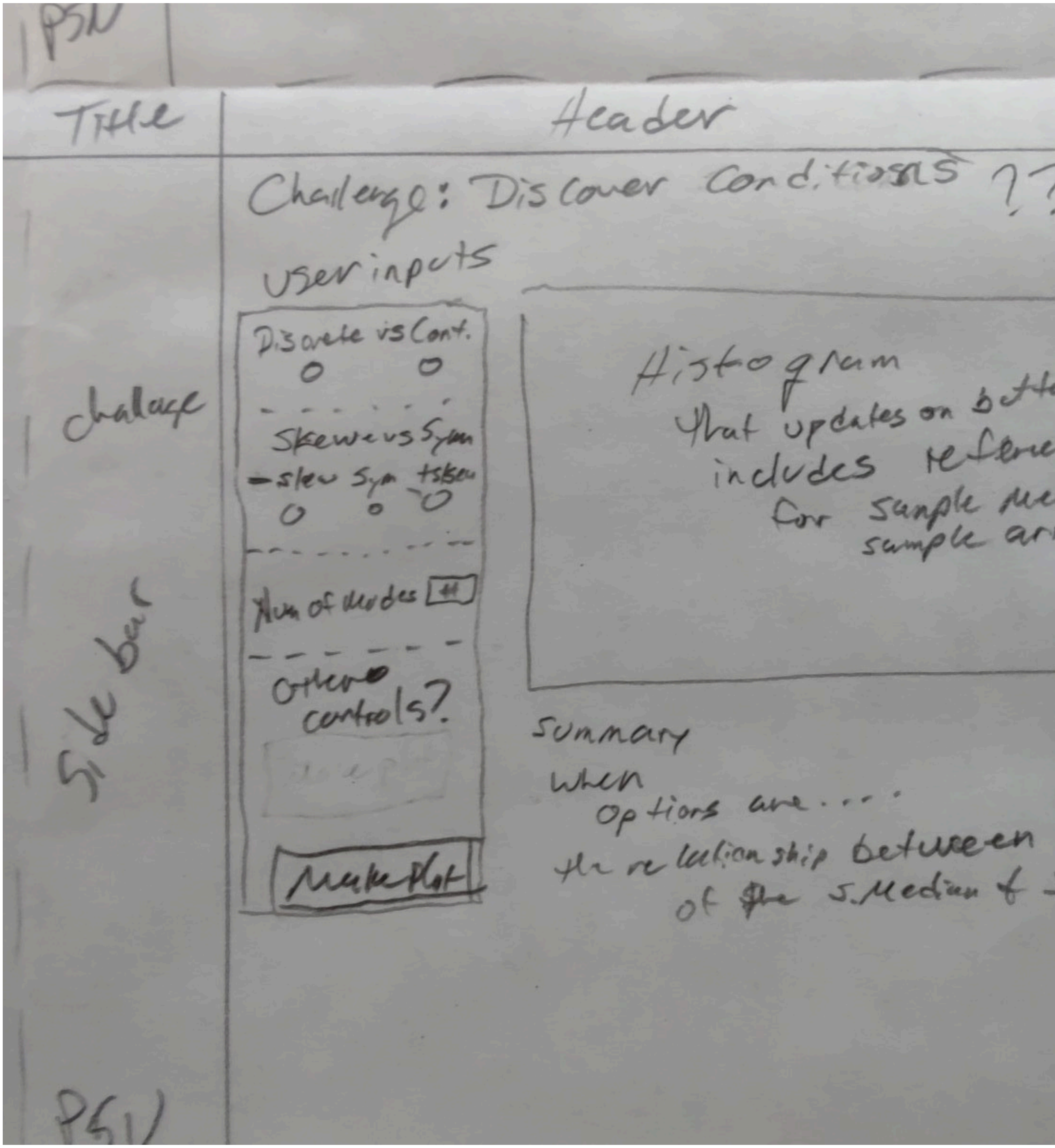


Figure 1.5: Fourth Quadrant: Challenge Tab

1.5 Steps 5-8: Write, Edit, Test, Debug

We recommend starting with a copy of the Sample App as the basis. This will give you a good skeleton to use for your new app.

(*Still under construction*)

1.6 Step 9: Larger Scale Testing

(*Still under construction*)

1.7 Steps 10-11: Pull Requestion and Tweaks

(*Still under construction*)

Chapter 2

Workflow

As you work on apps, both updating existing and developing new ones, moving through a workflow can help you organize yourself.

2.1 Revising Existing Apps

1. Read the Style Guide
2. Explore the existing apps in the book
3. When you identify an app you wish to work on,
   a. Go to that app's repository in GitHub
   b. Create new issues on GitHub to log both bugs and suggestions for improvements
   c. Create an issue specifically if the App currently does not abide by this Style Guide
   d. Optional–assign yourself to the issue
4. Download the current version of the app by using either RStudio or GitHub Desktop
5. Create a new branch for your developments
6. Begin editing the code
7. As you edit, be sure to reference this Style Guide and test your code locally via Run App in RStudio
8. Periodically push your edits to your development branch; don't forget to add commit messages and reference any issues
9. When you're ready to do larger scale testing you'll need to publish your App to the TLT RStudio Connect server (see Section 2.3)
10. When you've reached a point where you've finished editing, push your most recent commit to your development branch and then create a Pull Request; assign Neil as a reviewer
11. Make any requested changes and re-push to your development branch; update the Pull Request with a new comment

If everything checks out, then we'll merge your development branch with the master branch and schedule formal deployment of your App.

2.2 Creating New Apps

To see this workflow in action, check out Chapter 1.

1. Read the Style Guide.
2. Identify a topic for your new App.
3. Sketch out your plans for the App. **This should occur BEFORE you start coding.** Be sure that you include:
    a. Suggested Title
    b. Goal of the app
    c. What will the user be doing (i.e., potential inputs)
    d. What will the user be experiencing (i.e., potential outputs)
    e. Relationships between various elements.
4. When your plan **gets approval**, create a new repository on GitHub (see Section 2.2.1); you can initially load the Sample App template to this repository, if you wish.
5. Begin writing the code.
6. As you edit, be sure to reference this Style Guide and test your code locally via Run App in RStudio.
7. When you get the basic structure of your App set up, push your code to GitHub and create a new development branch for your continued editing. You can open new issues on your app as you go.
8. Periodically push your edits to your development branch. Don't forget to add commit messages and reference any issues.
9. When you're ready to do larger scale testing you'll need to publish your App to the TLT RStudio Connect server (see Section 2.3).
10. When you've reached a point where you've finished editing, push your most recent commit to your development branch and then create a Pull Request. Assign Neil as a reviewer.
11. Make any requested changes and re-push to your development branch. Update the Pull Request with a new comment.

If everything checks out, then we'll merge your development branch with the master branch and schedule formal deployment of your App.

2.2.1 GitHub Repo Names

Each Shiny App has its own repository (repo) on GitHub. As you begin to create new apps, you'll have to create a new repo on GitHub for each one. The name of that repo is extremely important as this will play a critical role in establishing the URL for your App. To this end, you need to adhere to following guidelines:

- Use Title Case (not camelCase)
- Use underscores (_) instead of spaces

- Match the App name as closely as possible
- You have a 100 character limit

While we can change repo names, doing so results in a large number of edits that have to be made. Thus, think carefully about how you are going to name your repository.

If you come across a repo that is poorly named, please make an issue and provide some suggestions for new names.

2.3 Testing Your App

To test your App beyond your local machine, we will be making use of TLT's RStudio Connect platform. You will need to follow these directions.

2.3.1 Set Up/Login to the VPN

Please refer to the following PSU Knowledge Basis Articles:

- VPN for Mac OS
- VPN for Windows
- VPN for Apple iOS
- VPN for Android

You MUST be logged into the PSU VPN to both upload and test your App on each device.

2.3.2 Connecting Your RStudio to TLT's RStudio Connect

You only need to do this step once.

1. After you've logged into the PSU VPN, go to TLT's RStudio Connect and log in using your PSU ID.
2. Once logged in, click on the Publish button as shown in the green box of Figure 2.1
3. Select Shiny App and a pop up window will appear.
4. Follow the steps in this window (especially Step 4) to set up the connection.

2.3.2.1 Checking You've Connected

You can check to see if you've connected by going into RStudio's Options and looking at the Publishing tab. You should see a similar entry as what is in the green box of Figure 2.2:

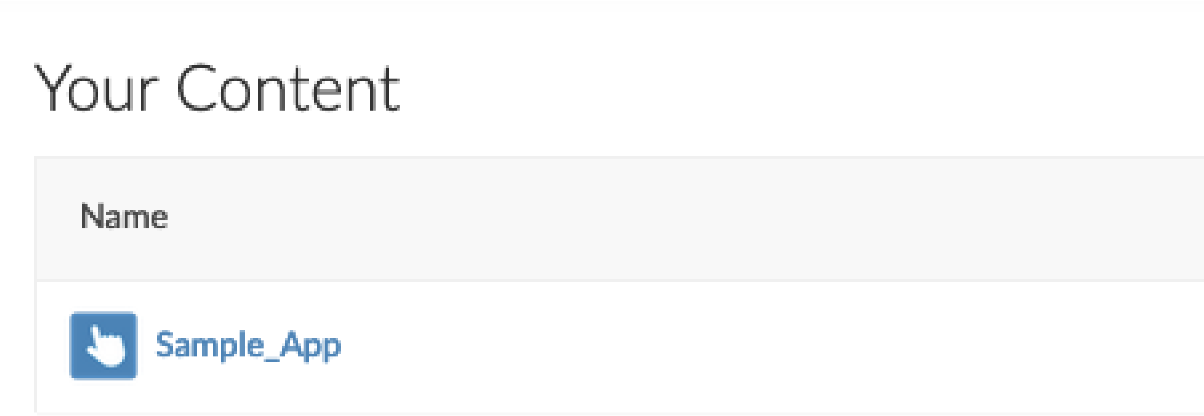


Figure 2.1: Click the Publish Button

2.3.3 Publishing Your App for Testing

Make sure that you are connected to PSU's VPN and that you've already connected your RStudio to TLT's server.

Click on the Publishing Icon, located just to the right of the Run App button. Be sure to select the TLT Server.

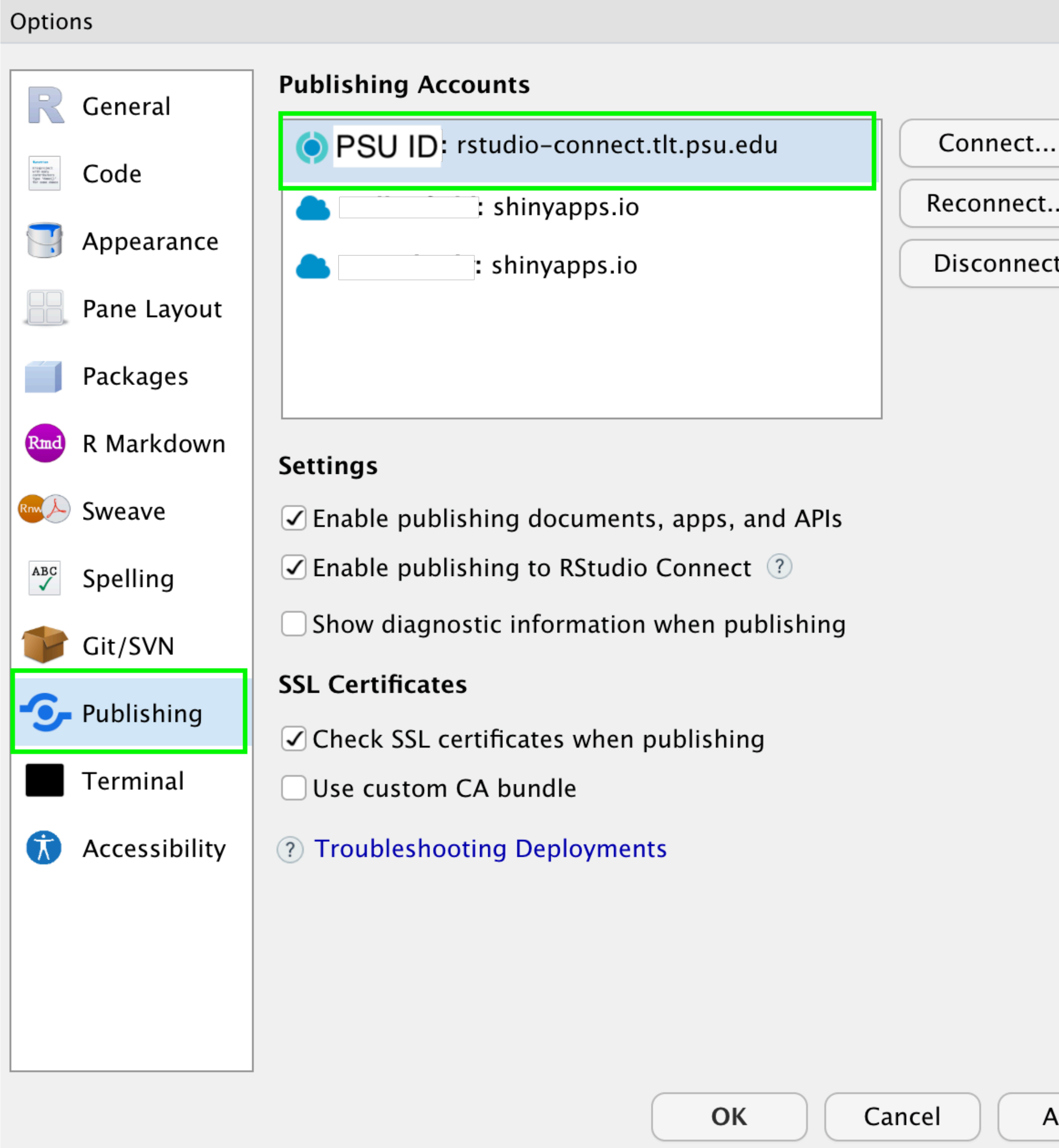
2.3.4 Configuring for Testing

Once your App has been published, a new window should open in your browser that shows your App plus the optional controls.

In Figure 2.3, you will need to change a setting to enable others (people and devices) to test your app. Click on the Access tab and set Who can view this application to Anyone-no login required. (These are highlighted in the green boxes.) Keep in mind that all users/devices MUST first connect to the PSU VPN to access your app.

If you want, you can add collaborators as I did in the example (orange box). This is not required.

The most important piece is the Content URL (marked with the blue star). You'll need to copy this URL and give that URL out to your testers. This will allow them easily access your app, regardless of the type of device they are using.



Publishing Accounts

PSU ID: rstudio-connect.tlt.psu.edu

[redacted]: shinyapps.io

[redacted]: shinyapps.io

Settings

☑ Enable publishing documents, apps, and APIs

☑ Enable publishing to RStudio Connect ?

☐ Show diagnostic information when publishing

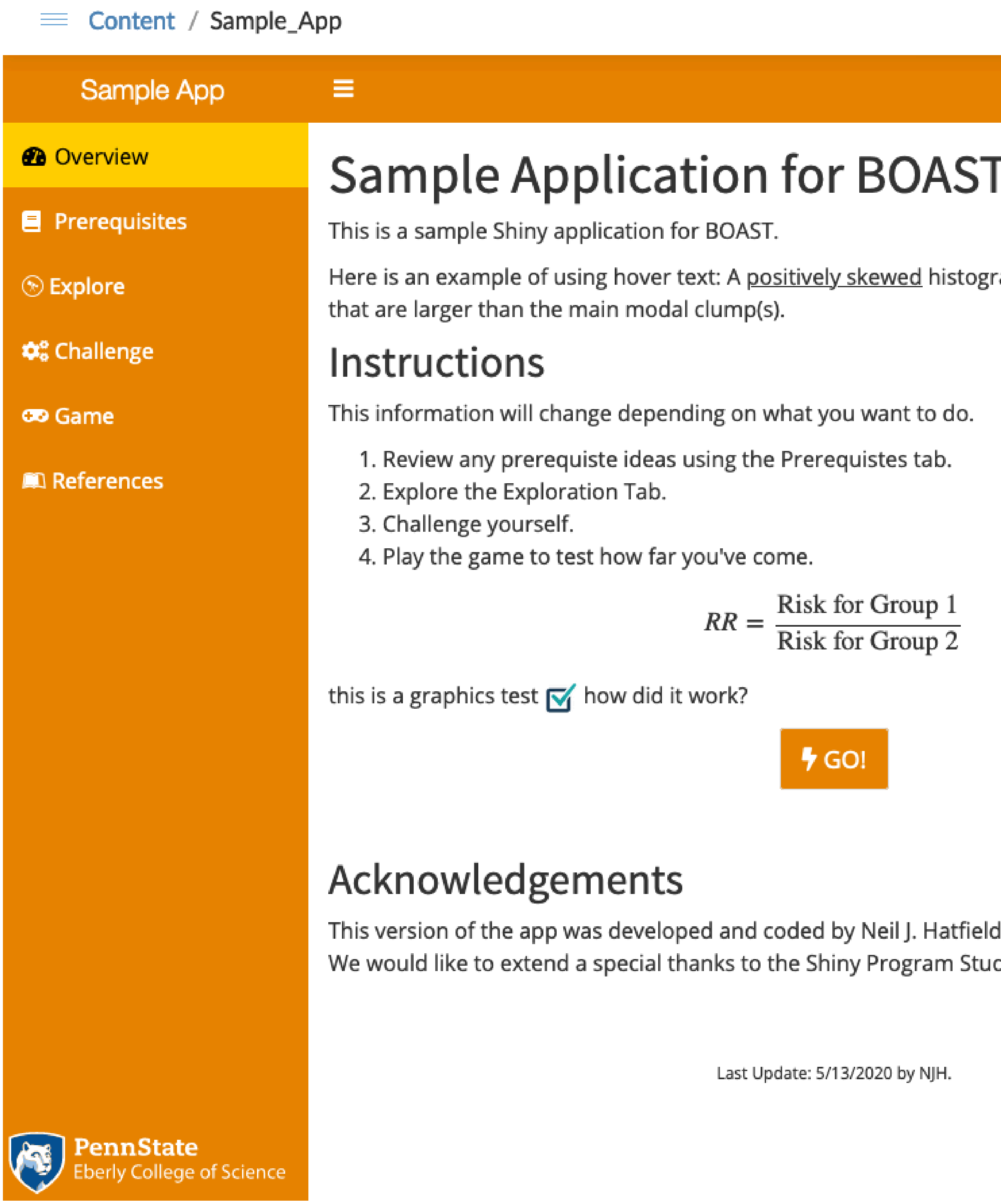
SSL Certificates

☑ Check SSL certificates when publishing

☐ Use custom CA bundle

? Troubleshooting Deployments

Figure 2.2: Check Your Publishing Profile



Sample Application for BOAST

This is a sample Shiny application for BOAST.

Here is an example of using hover text: A positively skewed histogra
that are larger than the main modal clump(s).

Instructions

This information will change depending on what you want to do.

1. Review any prerequiste ideas using the Prerequistes tab.
2. Explore the Exploration Tab.
3. Challenge yourself.
4. Play the game to test how far you've come.

$$RR = \frac{\text{Risk for Group 1}}{\text{Risk for Group 2}}$$

this is a graphics test ☑ how did it work?

Acknowledgements

This version of the app was developed and coded by Neil J. Hatfield
We would like to extend a special thanks to the Shiny Program Stuc

Last Update: 5/13/2020 by NJH.

Figure 2.3: Successful Publish and Settings

2.3.5 Check the Logs

As you test your App, you'll want to look for any error messages and/or warnings that get generated. Click on the Logs tab (red box in Figure 2.3) to view.

2.3.6 Problems?

If you run into problem either publishing your App or getting the App to launch on the TLT, please reach out to Neil and Bob.

Part II

Getting Ready to Code

Chapter 3

The Basics

Before you get too far into the Style Guide, we would like for you to take a moment and ensure that you have the following tools at your disposal.

3.1 Getting Started Checklist

1. Ensure that you have all of necessary accounts and if not, put in a request to Bob (rpc5102).

    a. DataCamp

    b. GitHub

    c. EducationShinyAppTeam
    d. BOAST in Microsoft Teams (tied to your PSU AccessID)

2. Ensure that you have all of the required software

    a. R (version 3.5.* minimum, version 4.0.0 preferred)
    b. RStudio Desktop (most current version preferred)

3. Additional Software that we strongly recommend

    a. GitHub Desktop

4. R Packages–here are some basic packages that everyone will need; be sure to install their dependencies

    ```
    install.packages(c("devtools", "DT", "ggplot2", "shiny", "shinyBS",
                       "shinyWidgets", "tidyverse", "tippy"))
    ```

5. A copy of the Sample App (see below)

3.2 `boastUtils` Package

Bob created the boastUtils package to automate much of the design and development process. This will not only reduce the amount of work you'll need to do, it'll also make apps more consistent.

Starting Summer 2020, you will be required to you make use of this tool.

Please check out the package's GitHub page for instructions on installing and usage.

```r
library(devtools)
devtools::install_github("EducationShinyAppTeam/boastUtils")
```

3.3 Sample App

Bob and Neil have created a Sample App repository that you can use as a template for your own apps. To get started, clone the Sample_App template repo found on GitHub. This will provide you with a skeleton for organizing your files as well as your code.

There are several methods you can use:

- Direct Download (Basic)
- GitHub Desktop (Recommended)
- Command Line (Advanced)
- RStudio Project

3.3.1 Direct Download

You can download this repository directly by visiting:

> https://github.com/EducationShinyAppTeam/Sample_App/archive/master.zip

3.3.2 GitHub Desktop

If you are using GitHub Desktop and have linked your account that has access to the EducationShinyAppTeam repository, you can do the following from inside GitHub Desktop:

1. Bring up the Clone Repository Menu (File -> Clone Repository…)

2. Enter Sample_App in the search bar and select the option that says Sample_App (not sampleapp)
3. Click the Choose… button for the local path (this is where you want to the clone to live on your computer)
4. Click the Clone button.

See also:

- GitHub Desktop Help

3.3.3 Command Line

Enter the following command in your terminal:

```
git clone git@github.com:EducationShinyAppTeam/Sample_App.git
```

See also:

- Git Handbook
- Resources to learn Git
- Happy Git and GitHub for the userR

3.3.4 RStudio Project

You can also use RStudio to connect to and copy repositories from GitHub. There are a few extra steps that you'll need to go through compared to other methods. We recommend that you use GitHub Desktop.

1. In your browser, navigate to the repository you want to copy; in this case the Sample_App repo.
2. Launch RStudio and create a new project.
3. Select the Version Control option and then Git.
4. Switch back to your browser and click on the Clone or Download button. There should be a line of text starting with `git@github` that you can copy. Do so.
5. Returning to RStudio, paste the get you just copied into the Repository URL field.
6. Adjust the directory name and the subdirectory (where this project will live on your machine) as you see fit.
7. Click the Create Project button.

This method will ensure that you will be able to successfully push and pull changes to GitHub. For additional help, Neil has created a video "Using RStudio to Connect to GitHub" which walks you through the process.

Chapter 4

The Basics

Before you get too far into the Style Guide, we would like for you to take a moment and ensure that you have the following tools at your disposal.

4.1 Getting Started Checklist

1. Ensure that you have all of necessary accounts and if not, put in a request to Bob (rpc5102).

   a. DataCamp

   b. GitHub

   c. EducationShinyAppTeam
   d. BOAST in Microsoft Teams (tied to your PSU AccessID)

2. Ensure that you have all of the required software

   a. `R` (version 3.5.* minimum, version 4.0.0 preferred)
   b. RStudio Desktop (most current version preferred)

3. Additional Software that we strongly recommend

   a. GitHub Desktop

4. `R` Packages–here are some basic packages that everyone will need; be sure to install their dependencies

```
install.packages(c("devtools", "DT", "ggplot2", "shiny", "shinyBS",
                   "shinyWidgets", "tidyverse", "tippy"))
```

5. A copy of the Sample App (see below)

4.2 boastUtils Package

Bob created the boastUtils package to automate much of the design and development process. This will not only reduce the amount of work you'll need to do, it'll also make apps more consistent.

Starting Summer 2020, you will be required to you make use of this tool.

Please check out the package's GitHub page for instructions on installing and usage.

```r
library(devtools)
devtools::install_github("EducationShinyAppTeam/boastUtils")
```

4.3 Sample App

Bob and Neil have created a Sample App repository that you can use as a template for your own apps. To get started, clone the Sample_App template repo found on GitHub. This will provide you with a skeleton for organizing your files as well as your code.

There are several methods you can use:

- Direct Download (Basic)
- GitHub Desktop (Recommended)
- Command Line (Advanced)
- RStudio Project

4.3.1 Direct Download

You can download this repository directly by visiting:

> https://github.com/EducationShinyAppTeam/Sample_App/archive/master.zip

4.3.2 GitHub Desktop

If you are using GitHub Desktop and have linked your account that has access to the EducationShinyAppTeam repository, you can do the following from inside GitHub Desktop:

1. Bring up the Clone Repository Menu (File -> Clone Repository…)

2. Enter Sample_App in the search bar and select the option that says Sample_App (not sampleapp)
3. Click the Choose… button for the local path (this is where you want to the clone to live on your computer)
4. Click the Clone button.

See also:

- GitHub Desktop Help

4.3.3 Command Line

Enter the following command in your terminal:

```
git clone git@github.com:EducationShinyAppTeam/Sample_App.git
```

See also:

- Git Handbook
- Resources to learn Git
- Happy Git and GitHub for the userR

4.3.4 RStudio Project

You can also use RStudio to connect to and copy repositories from GitHub. There are a few extra steps that you'll need to go through compared to other methods. We recommend that you use GitHub Desktop.

1. In your browser, navigate to the repository you want to copy; in this case the Sample_App repo.
2. Launch RStudio and create a new project.
3. Select the Version Control option and then Git.
4. Switch back to your browser and click on the Clone or Download button. There should be a line of text starting with `git@github` that you can copy. Do so.
5. Returning to RStudio, paste the get you just copied into the Repository URL field.
6. Adjust the directory name and the subdirectory (where this project will live on your machine) as you see fit.
7. Click the Create Project button.

This method will ensure that you will be able to successfully push and pull changes to GitHub. For additional help, Neil has created a video "Using RStudio to Connect to GitHub" which walks you through the process.

Part III

Style Guide-Coding

Chapter 5

Coding Style

Now that you've gone through the Getting Started chapters, we can turn our attention to the first part of the Style Guide: Coding. There are many aspects that fall under the heading of Coding Style.

5.1 General Coding Style

Our general practice is to make use of the tidyverse style guide. However, we do have some important departures and additional practices you need to follow:

- Leave comments
- Use informative names
- Format your code
- Be explicit

5.1.1 Leave Comments

At bare minimum, use a comment to break your code into sections. This helps you and others conceptualize the code into more manageable chunks. Your comments can provide others with potential keywords to search for when looking at your code later on.

For particularly complex sections, use comments to summarize what you're trying to do. This can help you and others pick up the coding thread for what you are trying to do for troubleshooting, debugging, and future improvements.

5.1.2 Informative Names

Use informative names for variables and functions in your code. Use names that give another person a sense of what that variable represents (nouns) or what the function is supposed to do (action verbs). For example,

- `scoreMatrix` is a matrix that holds a set of scores
- `checkGame` is a function that checks the state of the current game

Use camelCase for variable names and functions. The first word begins with a lowercase letter and additional words start with Capital letters with no spaces or underscores (_) between them. (This is a departure from the tidyverse style.)

Avoid using the variable names from code that you're making use of from another app or script. For example, don't use `waitTimes` to hold your data on the number of correct answers a user has given. It is also good practice to avoid re-using function names that appear in other libraries that are currently loaded to avoid namespace collision.

5.1.3 Format Your Code

See also Section 5.3

Use indentation spacing to help make your code readable. RStudio has a built-in tool that can help with this.

1. Select all of the code you want to reformat.
   - To select all code in a file, use Command-A (Mac) or Control-A (Windows).
2. Press Command-Shift-A (Mac), Control-Shift-A (Windows), or click on the Code menu and select Reformat Code.

NOTE: this tool is imperfect and can result in left parenthesis or curly braces moving up a line to where you might have an end-of-line comment, resulting in errors.

Additionally, you can make use of the `styler` and `lintr` packages to help you perform formatting checks and to quickly reformat code.

5.1.4 Be Explicit

One of the best practices a coder can do is to be explicit. And no, we don't mean that type of explicit. This links back to Informative Naming but goes a step beyond. `R` is a functional programming language. One of the important aspects of this is that functions in `R` obey the same rules as mathematical functions, especially multivariate functions.

One implicit fact about functions that most students don't realize is that the order of a function's arguments are a matter of convention and are actually arbitrary. While we might define f as $f(x, y) = x^2 + 3y$, we could call $f(y = 2, x = 1)$ and get the same output as $f(1, 2)$ when we use the convention set up in the definition. This issue is exacerbated with functions in `R`.

Therefore, when you pass values to the arguments of a function in `R`, you should be explicit and include the argument name in your code. For example,

```
actionButton(
  inputId = "submit",
  label = "Submit",
  color = "primary",
  style = "bordered",
  class = "btn-ttt"
)
```

There are a few exceptions to this:

- Formula Arguments (e.g., `response ~ factor1 + block`)
- Data Vectors/Frames (e.g., `dataFrame$response`)
- Functions from `shiny`, `shinydashboard`, `shinyBS`, etc.
  - E.g., you can call `dashboardHeader` rather than `shinydashboard::dashboardHeader`

General Advice:

- If you are calling a function from a package that is unique to your App or not commonly used, list the package in the function call.
- If you are using a common/central function (i.e., those from `shiny`, `ggplot2`), you can omit the package in the function call.

Remember the following: the more proactive you are from the get go in commenting and organizing your code, the easier time you will have for debugging and improving your code down the road.

5.2 Additional Coding Style

In addition, we use the following coding practices.

5.2.1 Minimize Package Usage

Make sure that you absolutely have to use a particular package before you do. Check to see if what you're trying to do can be done in a package you're already using or in base R.

This is not to say that you can't make use of a new package. This guideline's purpose is to streamline the various packages that get used in BOAST and to use the most of the packages that we are using. However, there are times when we just need to use a new package. Please try to use packages that are housed on CRAN and are preferably under active development. Searching GitHub for the package name can help you decide whether the package is active.

To help you avoid name masking (i.e., Section 5.1.4), ensure that you are actually using a package, and to help future readers follow your code, explicitly call

packages with their functions. For example, use `dplyr::filter([arguments])` instead of just `filter([arguments])`.

You can also run a `funchir::stale_package_check` on your `app.R`, `ui.R`, and `sever.R` files to see which packages you're loading but might not actually use in your code. These stale packages may then be deleted out from your library call.

```r
# Check working directory
getwd()
# Does the output match the folder path to where you app lives?
# If not, then you need to set that. For example,
setwd("~/Documents/GitHub/shiny-apps/Sample_App")

# Run a stale package check on app.R, ui.R, and server.R
funchir::stale_package_check("app.R")
```

Be aware that while `stale_package_check` is useful, it doesn't always catch everything. For example, when we ran it on the sample app, we were told that there were no exported functions from `ggplot2` or `tippy`. However, we know that there are functions from both of those. If you have a giant list of libraries to check, there might some more misses.

Once you're sure that a package isn't being used, remove the `library` call for that package from your code.

If you are dealing with a `ui.R`/`server.R` pair, make sure that the appropriate packages are loaded in the right file. For example, `ggplot2` only needs to be loaded in the `server.R` file.

5.2.2 Minimize External Files

Try to minimize the number and size of external files you're attaching to your App. If you're working on an existing app, remove any external files that are no longer necessary.

Wherever possible try to place external files into the `www` directory/folder that is at the same level as your `app.R` or `ui.R`/`server.R` files.

If there are any external files (e.g., `*.csv`, `*.txt`, `*.dat`, `*.jpg`) that are not being used, delete them from the repository.

5.2.3 Plot Caching

There are cases when your app will include a plot that falls into any of the following categories:

1) a plot that all users will see (i.e., static data set),
2) is a computationally intense plot (i.e., lots of data and/or layers),

3) might be a dynamic graph that the user explores and needs to move back and forth between various states.

Each of these cases represents a place where the performance of your app can suffer. To improve performance, you should consider using the `renderCachePlot` function rather than `renderPlot`. This function will store a copy of each plot on the sever and provide that stored copy to new instances of the app. This cuts down on server demands and speeds up your app. For the third category, this allows the users to move more quickly to a previously examined state (i.e., low to no hang time).

5.2.4 Styling Text

The styling of text (including font size, font weight (i.e., boldface), font family, color, etc.) is something that we began standardizing in Fall 2019. This is centrally managed by Neil and Bob to ensure 1) consistency across apps, 2) to cut down on the extraneous coding you need to do (this way you can focus on the apps), and 3) ensures that our apps are accessible and mobile friendly. To this end:

The styling of text will be controlled using an **external** CSS (Cascading Style Sheet) file. You will implement this depending on what approach you use:

- If you're using the `boastApp` function from `boastUtils` (**recommended for all new apps**), you don't need to do anything as the appropriate calls will automatically be done for you.
- If you are using the `ui.R` and `server.R` format (i.e., apps from prior years), place the following code in the `tags$head( )` portion at the top of the `dashboardBody` section of the `ui.R` file :

```
dashboardPage(
  skin = "blue",
  dashboardHeader(
    title = "title",
    titleWidth = 300,
    # [omitted code]
  ),
  # [omitted code],
  dashboardBody(
    tags$head(
      tags$link(rel = "stylesheet", type = "text/css",
        href = "https://educationshinyappteam.github.io/Style_Guide/theme/boast.css"),
      tabItems(
       # [omitted code]
      )  )  )  )
```

There times when you might need some additional text styling that is not already defined. In these cases, **you'll need to talk to Neil or Bob to determine**

what is the best course of action. This could entail an addition to the central CSS file or the inclusion of an additional CSS file unique to your App.

The central CSS file should only be altered by Neil or Bob and covers many aspects. However, this is a living file meaning that we will add to/alter the file as needed to improve the whole set of BOAST apps.

See also Section 5.5.

5.3 Organizing Your Code

There is a fixed order in which you should write your code. This will depend on if you are using a singular `app.R` file or the pair `ui.r` and `server.r`.

5.3.1 Using `app.R`

The order for your code will be as follows:

1. Packages to be loaded
2. App Metadata (see below)
3. Any additional source files and custom functions
4. UI definition (see Chapter 8)
5. Server definition
6. `boastApp` call

5.3.2 Using `ui.R` and `server.R`

The order for your code in the `ui.R` file:

1. Packages to be loaded
2. App Metadata (see below)
3. Any additional source files and custom functions
4. UI definition (see Chapter 8)

The order for your code in the `server.R` file:

1. Packages to be loaded
2. Any additional source files and custom functions
3. Server definition

5.4 HTML

Our Shiny apps live on the internet and are thus going to be wrapped in HTML even though we are writing them in `R`. When you run a Shiny app, `R` and the `shiny` packages actually convert all of the `R` code into an HTML document which is served up to the user. While you will not have to directly write HTML for your apps, you should become aware of HTML tags that you will need to

deal with when coding/writing your App. Further, you need to learn how to use these tags correctly as failure to do so becomes an **Accessibility Issue**.

HTML Tags should not be used without some basic understanding of what that tag is for. Using the tags without such an understanding can not only lead to problems with your code running properly or looking like what you intended, but can interfere with how accessible your app is to a wider audience. Here are the most common HTML tags that you'll encounter when making a Shiny App.

5.4.1 Lists

There are two aspects to lists: the items in the list and the environment those items live in.

The two list environments are Ordered Lists and Unordered Lists.

- The Ordered List environment is for when you require that a user works through the items in a particular order. You call this environment in your App by using `tags$ol( [your list] )`. These lists are sequentially numbered/lettered.
- The Unordered List environment is for when you want to give the user a list where they can jump around between the items however they wish. You call this environment in your App by using `tags$ul( [your list] )`. These lists will appear with bullets.

Once you've created the list environment, you create items for your list in the same way: `tags$li([content])`. You do not need to and should **NOT** use any header or paragraph tags with your list items; our Style Sheet will take care of the formatting. You can use emphasis/italics or strong/boldface on portions of the content as appropriate.

Here is an example of how you might create a list:

```
tags$ul(
  tags$li("This is my first item."),
  tags$li("A second item with a", tags$strong("boldface"), "word.")
)
```

which turns into

- This is my first item.
- A second item with a **boldface** word.

There is one exception to the environment that you need to be aware of: the Dashboard Header has a built-in listing environment and thus you will jump straight to `tags$li()` when in that section.

5.4.2 Headers

Heading tags provide a navigational structure to your app. Think of them as being the different levels of titles in a book. In fact, if you look at the Table of Contents to the left (provided you haven't hidden it), everything you see there is actually tied to a Header in this document.

No matter how much new coders want, Heading Tags are **NOT** for making text larger, boldface, or other text styling. Think about the headings as laying out a Table of Contents for your app, rather than containing content. Just like the Table of Contents for this Style Guide.

There is a specific ordering to the Header tags that is critical to ensure your App is accessible by screen readers.

1. `h1()` is for the Title of your App and should be ONCE at the top of the first page that appears when you load the app (i.e., the Overview).
2. `h2()` is for Tab/Page titles within the App. These would correspond with the tab links you place in the dashboard's left side panel.
3. `h3()` is for titling Sections within a Tab/Page of the App. These might title the portion of the page that is for a game board, questions, answers, and graphs/plots.
4. `h4()` is for a Subsection within a section. You might use this to distinguish different sets of controls in a Controls section.
5. `h5()` and `h6()` should be used sparingly. These might be used for different levels of a game. When you call these in your App, you just call them as listed here; i.e., `h1()`, `h2()`, etc.

Avoid skipping heading levels as this will get your App flagged for an Accessibility Violation. That is to say don't start at `h2` with no `h1` and don't go from `h1` to `h3` without an `h2`. Again, think of these as the layers of your table of contents or the outline of a paper; you wouldn't skip whole sections in either of those.

Here are few more things to NOT do with the heading tags:

- DO NOT USE headings to style text (We cannot stress this enough.)
- Do not wrap a header tag around a list element (e.g., `h3(tags$li("here is my list item"))`) nor the reverse (e.g., `tags$li(h3("here is my list item"))`)
- Do not mix header tags together in the same line or with the paragraph tag (e.g., 'h2("Introduction to", p("my game")))

For more information check out

- W3C's Tutorial
- Penn State's Headings and Subheadings Accessibility

5.4.3 Paragraphs

Your content should be enclosed in the paragraph tag, `p()`. Even if your content is relatively brief or a mathematical expression, the paragraph tag will ensure that the content is sized correctly and readable. This tag tells screen readers and browsers what your content actually is.

5.5 CSS

Cascading Style Sheets (CSS) will be the way to control the visual appearance of all elements of the app. To ensure that we have consistent we will make use of the BOAST CSS throughout. This is relatively new (Winter 2019/Spring 2020), so many of the older apps will need to have this added to their code. This allows us to centralize and dynamically update all apps at once.

There are a few key elements at play for CSS:

1. All apps need to have the appropriate reference to the CSS file (See Section 5.2.4).
    a. Please register any issues, bugs, enhancements, new styling to the Style Guide repository of GitHub.
2. Any App specific styling needs to as be in a CSS file and approved.
3. There should not be any styling code in the `app.R`, `ui.R`, or `server.R` files (a.k.a. in-line CSS). Styling code would look like this:
    - `tags$style(HTML(".js-irs-0 .irs-single, .js-irs-0 .irs-bar-edge, .js-irs-0 .irs-bar {background: orange}"))`
    - `style="color: #fff; background-color: #337ab7; border-color: #2e6da4"`
4. For objects that need to have a particular styling invoked, we will make use of the `class` attribute. These will invoked through the HTML tags or through the function calls.

```r
# HTML Tag Example 1
p(
  class = "hangingindent",
  "Attali, D. and Edwards, T. (2018). shinyalert: Easily create
  pretty popup messages (modals) in 'Shiny'. (v1.0). [R package].
  Available from https://CRAN.R-project.org/package=shinyalert"
)

# HTML Tag Example 2
div(class = "updated", "Last Update: 12/2/2019 by NJH.")

# Function Call Example
actionButton(
  inputId = paste0("grid-", row, "-", column),
  label = "?",
  color = "primary",
  style = "bordered",
  class = "grid-fill"
)
```

There is an exception to #3: setting up alignment for a div section. This would look like `div(style = "text-align: center"...)` This type of styling is allowed in the R files (generally in the UI portions only).

If you come across an app that has in-line CSS OR calls to a CSS file that isn't `boast.css`, please log an issue and mention/assign Neil. (Most common external CSS files are `style.css` and `Feature(s).css`.)

5.6 Metadata

As of boastUtils `v0.1.10`, your App will need the following metadata in a file named DESCRIPTION (no file extension). This data is used to inform Learning Record Stores (LRS) about your App as well as provide information to instructors using the Instructor App (unreleased).

DESCRIPTION

```
Title: Sample App - A Lengthy Title
ShortName: Sample App
Date: 2020-12-31
Authors@R: c(
    person("Lorem", "Ipsum", email = "lorem.ipsum@psu.edu", role = c("aut", "cre")),
    person("Ang", "Li", email = "ang.li@psu.edu", role = c("ctb"))
  )
Chapter: Sample Chapter
Description: This app is focused on the common types of xyz.
LearningObjectives: c(
    "The student will learn to understand Concept A in way z.",
    "The student will learn to understand Concept B as description y"
  )
DisplayMode: Normal
URL: https://psu-eberly.shinyapps.io/Sample_App
BugReports: https://github.com/EducationShinyAppTeam/Sample_App/issues
License: CC-BY-NC-SA-4.0
Tags: simulation
Type: Shiny
```

5.7 Using rLocker

Because these apps are being developed for educational purposes, it is beneficial to collect data (logfiles) based on learning objectives and outcomes. The rlocker package was created to aid in this process. It is important to know that the package itself does not perform feats of magic and will require some tuning to get right.

5.7.1 Installing

```r
library(devtools)
devtools::install_github("rpc5102/rlocker")
```

View Documentation

5.7.2 Setup

Core configuration is included in the boastUtils package. Simply use the boastApp wrapper function instead of shinyApp and you're done! See Creating an App for more information on how to use boastApp.

5.7.3 Usage

The main purpose of this package is to help create meaningful- structured- xAPI data. xAPI (Experience API) can be thought of as the "Who did what, where did they do it, and when did they do it?" in your App. Statements are often structured as Actor, Verb, and Object that can also Result in something. For more on xAPI, check out What is the Experience API? or experiment with Statements in the xAPI Lab.

For example:

> **Bob** (Actor) **clicked** (Verb) **submit** (Object).

In assessments:

> **Neil** (Actor) **answered** (Verb) **Question 1** (Object) **correctly** with the answer **true** (Result).

This is a good way to think about it when beginning to write collection functions. Which brings us to our next part, writing collection functions.

Out of the box, `rlocker` does not provide any collection functions for apps, only creation and storage mechanisms. Why is this? Every app is different! You know your app the best and what interactions are possible. Before storing data in a Learning Record Store (LRS) like Learning Locker, it is important to transform it in a way that makes sense.

boastUtils

If using the boastApp wrapper for your project, your App will automatically generate and store a few of the more typical Statements. Please refrain from creating additional versions of the following statement descriptions:

Statement	Description
launched	User has started the app.
experienced	User has visited a tab (page) within the app.
exited *	User has left the app.

An App should only have one launched and exited event but can have multiple experiences per session. Therefore, you may use experienced in other places within your App.

* Requires boastUtils >= 0.1.5.

Sample generator function

```
.generateStatement <- function(
  session,
  verb = NA,
  object = NA,
  description = NA,
  interactionType = NA,
  response = NA,
  success = NA,
  completion = FALSE)
{
  statement <- rlocker::createStatement(list(
    verb = verb,
    object = list(
      id = paste0(getCurrentAddress(session), "#", object),
      name = paste0(APP_TITLE),
      description = paste0("Question ", activeQuestion, ": ", description),
      interactionType = interactionType
    ),
    result = list(
      success = success,
      response = response,
      completion = completion
    )
  ))
  return(rlocker::store(session, statement))
}
```

Sample event observer

```
observeEvent(session$input[[id]], {

  # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
  # BEGIN VALIDATION                                                          #
  #                                                                           #
  # For this example,                                                         #
  #   We will check if someone answered a question in our game correctly.     #
  #   Your code should be different here!                                     #
  #                                                                           #

  # The input element we're observing.                                        #
  object <- id

  # The user's action verb. Most likely going to be answered in this case.    #
  # Run rlocker::getVerbList() if you are unsure of the options.              #
  verb <- "answered"

  # The correct answer to the question.                                       #
  answer <- gameSet[id, "answer"]

  # The user's answer to the question.                                        #
  response <- input$ans

  # A description of the question or the full question itself (if short).    #
  description <- gameSet[id,]$question

  # The type of question it is.                                               #
  # Run rlocker::getInteractionTypes() if you are unsure of the options.      #
  interactionType <- ifelse(
    gameSet[id,]$format == "numeric", "numeric", "choice"
  )

  # Was the question answered successfully?                                   #
  success <- input$ans == answer

  # Did this event trigger the completion of your activity?                   #
  completion <- ifelse(.appState == "continue", FALSE, TRUE)

  #                                                                           #
  # END VALIDATION                                                            #
  # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

  .generateStatement(
    session,
    object = object,
    verb = verb,
    description = description,
    response = response,
    interactionType = interactionType,
    success = success,
    completion = completion
  )
}
```

Additional code can be found in the examples folder as well as the main README of the rLocker project; apps such as the Hypothesis_Testing_Game have already been outfitted with rlocker. The statement generator above will eventually make its way into `boastUtils` once enough feedback is collected. If you are ever confused about how it works, feel free to reach out to Bob (rpc5102).

Did you know?

You can run `help(package = "rlocker")` to view the help files for this package or press `F1` while typing a function name to see the documentation for that specific function.

Chapter 6

Coding Style

Now that you've gone through the Getting Started chapters, we can turn our attention to the first part of the Style Guide: Coding. There are many aspects that fall under the heading of Coding Style.

6.1 General Coding Style

Our general practice is to make use of the tidyverse style guide. However, we do have some important departures and additional practices you need to follow:

- Leave comments
- Use informative names
- Format your code
- Be explicit

6.1.1 Leave Comments

At bare minimum, use a comment to break your code into sections. This helps you and others conceptualize the code into more manageable chunks. Your comments can provide others with potential keywords to search for when looking at your code later on.

For particularly complex sections, use comments to summarize what you're trying to do. This can help you and others pick up the coding thread for what you are trying to do for troubleshooting, debugging, and future improvements.

6.1.2 Informative Names

Use informative names for variables and functions in your code. Use names that give another person a sense of what that variable represents (nouns) or what the function is supposed to do (action verbs). For example,

- `scoreMatrix` is a matrix that holds a set of scores
- `checkGame` is a function that checks the state of the current game

Use camelCase for variable names and functions. The first word begins with a lowercase letter and additional words start with Capital letters with no spaces or underscores (_) between them. (This is a departure from the tidyverse style.)

Avoid using the variable names from code that you're making use of from another app or script. For example, don't use `waitTimes` to hold your data on the number of correct answers a user has given. It is also good practice to avoid re-using function names that appear in other libraries that are currently loaded to avoid namespace collision.

6.1.3 Format Your Code

See also Section 5.3

Use indentation spacing to help make your code readable. RStudio has a built-in tool that can help with this.

1. Select all of the code you want to reformat.
   - To select all code in a file, use Command-A (Mac) or Control-A (Windows).
2. Press Command-Shift-A (Mac), Control-Shift-A (Windows), or click on the Code menu and select Reformat Code.

NOTE: this tool is imperfect and can result in left parenthesis or curly braces moving up a line to where you might have an end-of-line comment, resulting in errors.

Additionally, you can make use of the `styler` and `lintr` packages to help you perform formatting checks and to quickly reformat code.

6.1.4 Be Explicit

One of the best practices a coder can do is to be explicit. And no, we don't mean that type of explicit. This links back to Informative Naming but goes a step beyond. `R` is a functional programming language. One of the important aspects of this is that functions in `R` obey the same rules as mathematical functions, especially multivariate functions.

One implicit fact about functions that most students don't realize is that the order of a function's arguments are a matter of convention and are actually arbitrary. While we might define f as $f(x, y) = x^2 + 3y$, we could call $f(y = 2, x = 1)$ and get the same output as $f(1, 2)$ when we use the convention set up in the definition. This issue is exacerbated with functions in `R`.

Therefore, when you pass values to the arguments of a function in R, you should be explicit and include the argument name in your code. For example,

```
actionButton(
  inputId = "submit",
  label = "Submit",
  color = "primary",
  style = "bordered",
  class = "btn-ttt"
)
```

There are a few exceptions to this:

- Formula Arguments (e.g., `response ~ factor1 + block`)
- Data Vectors/Frames (e.g., `dataFrame$response`)
- Functions from `shiny`, `shinydashboard`, `shinyBS`, etc.
  - E.g., you can call `dashboardHeader` rather than `shinydashboard::dashboardHeader`

General Advice:

- If you are calling a function from a package that is unique to your App or not commonly used, list the package in the function call.
- If you are using a common/central function (i.e., those from `shiny`, `ggplot2`), you can omit the package in the function call.

Remember the following: the more proactive you are from the get go in commenting and organizing your code, the easier time you will have for debugging and improving your code down the road.

6.2 Additional Coding Style

In addition, we use the following coding practices.

6.2.1 Minimize Package Usage

Make sure that you absolutely have to use a particular package before you do. Check to see if what you're trying to do can be done in a package you're already using or in base R.

This is not to say that you can't make use of a new package. This guideline's purpose is to streamline the various packages that get used in BOAST and to use the most of the packages that we are using. However, there are times when we just need to use a new package. Please try to use packages that are housed on CRAN and are preferably under active development. Searching GitHub for the package name can help you decide whether the package is active.

To help you avoid name masking (i.e., Section 5.1.4), ensure that you are actually using a package, and to help future readers follow your code, explicitly call

packages with their functions. For example, use `dplyr::filter([arguments])` instead of just `filter([arguments])`.

You can also run a `funchir::stale_package_check` on your `app.R`, `ui.R`, and `sever.R` files to see which packages you're loading but might not actually use in your code. These stale packages may then be deleted out from your library call.

```
# Check working directory
getwd()
# Does the output match the folder path to where you app lives?
# If not, then you need to set that. For example,
setwd("~/Documents/GitHub/shiny-apps/Sample_App")

# Run a stale package check on app.R, ui.R, and server.R
funchir::stale_package_check("app.R")
```

Be aware that while `stale_package_check` is useful, it doesn't always catch everything. For example, when we ran it on the sample app, we were told that there were no exported functions from `ggplot2` or `tippy`. However, we know that there are functions from both of those. If you have a giant list of libraries to check, there might some more misses.

Once you're sure that a package isn't being used, remove the `library` call for that package from your code.

If you are dealing with a `ui.R`/`server.R` pair, make sure that the appropriate packages are loaded in the right file. For example, `ggplot2` only needs to be loaded in the `server.R` file.

6.2.2 Minimize External Files

Try to minimize the number and size of external files you're attaching to your App. If you're working on an existing app, remove any external files that are no longer necessary.

Wherever possible try to place external files into the `www` directory/folder that is at the same level as your `app.R` or `ui.R`/`server.R` files.

If there are any external files (e.g., `*.csv`, `*.txt`, `*.dat`, `*.jpg`) that are not being used, delete them from the repository.

6.2.3 Plot Caching

There are cases when your app will include a plot that falls into any of the following categories:

1) a plot that all users will see (i.e., static data set),
2) is a computationally intense plot (i.e., lots of data and/or layers),

3) might be a dynamic graph that the user explores and needs to move back and forth between various states.

Each of these cases represents a place where the performance of your app can suffer. To improve performance, you should consider using the `renderCachePlot` function rather than `renderPlot`. This function will store a copy of each plot on the sever and provide that stored copy to new instances of the app. This cuts down on server demands and speeds up your app. For the third category, this allows the users to move more quickly to a previously examined state (i.e., low to no hang time).

6.2.4 Styling Text

The styling of text (including font size, font weight (i.e., boldface), font family, color, etc.) is something that we began standardizing in Fall 2019. This is centrally managed by Neil and Bob to ensure 1) consistency across apps, 2) to cut down on the extraneous coding you need to do (this way you can focus on the apps), and 3) ensures that our apps are accessible and mobile friendly. To this end:

The styling of text will be controlled using an **external** CSS (Cascading Style Sheet) file. You will implement this depending on what approach you use:

- If you're using the `boastApp` function from `boastUtils` (**recommended for all new apps**), you don't need to do anything as the appropriate calls will automatically be done for you.
- If you are using the `ui.R` and `server.R` format (i.e., apps from prior years), place the following code in the `tags$head( )` portion at the top of the `dashboardBody` section of the `ui.R` file :

```
dashboardPage(
  skin = "blue",
  dashboardHeader(
    title = "title",
    titleWidth = 300,
    # [omitted code]
  ),
  # [omitted code],
  dashboardBody(
    tags$head(
      tags$link(rel = "stylesheet", type = "text/css",
        href = "https://educationshinyappteam.github.io/Style_Guide/theme/boast.css"),
      tabItems(
       # [omitted code]
      )  )  )  )
```

There times when you might need some additional text styling that is not already defined. In these cases, **you'll need to talk to Neil or Bob to determine**

what is the best course of action. This could entail an addition to the central CSS file or the inclusion of an additional CSS file unique to your App.

The central CSS file should only be altered by Neil or Bob and covers many aspects. However, this is a living file meaning that we will add to/alter the file as needed to improve the whole set of BOAST apps.

See also Section 5.5.

6.3 Organizing Your Code

There is a fixed order in which you should write your code. This will depend on if you are using a singular `app.R` file or the pair `ui.r` and `server.r`.

6.3.1 Using `app.R`

The order for your code will be as follows:

1. Packages to be loaded
2. App Metadata (see below)
3. Any additional source files and custom functions
4. UI definition (see Chapter 8)
5. Server definition
6. `boastApp` call

6.3.2 Using `ui.R` and `server.R`

The order for your code in the `ui.R` file:

1. Packages to be loaded
2. App Metadata (see below)
3. Any additional source files and custom functions
4. UI definition (see Chapter 9)

The order for your code in the `server.R` file:

1. Packages to be loaded
2. Any additional source files and custom functions
3. Server definition

6.4 HTML

Our Shiny apps live on the internet and are thus going to be wrapped in HTML even though we are writing them in `R`. When you run a Shiny app, `R` and the `shiny` packages actually convert all of the `R` code into an HTML document which is served up to the user. While you will not have to directly write HTML for your apps, you should become aware of HTML tags that you will need to

deal with when coding/writing your App. Further, you need to learn how to use these tags correctly as failure to do so becomes an **Accessibility Issue**.

HTML Tags should not be used without some basic understanding of what that tag is for. Using the tags without such an understanding can not only lead to problems with your code running properly or looking like what you intended, but can interfere with how accessible your app is to a wider audience. Here are the most common HTML tags that you'll encounter when making a Shiny App.

6.4.1 Lists

There are two aspects to lists: the items in the list and the environment those items live in.

The two list environments are Ordered Lists and Unordered Lists.

- The Ordered List environment is for when you require that a user works through the items in a particular order. You call this environment in your App by using `tags$ol( [your list] )`. These lists are sequentially numbered/lettered.
- The Unordered List environment is for when you want to give the user a list where they can jump around between the items however they wish. You call this environment in your App by using `tags$ul( [your list] )`. These lists will appear with bullets.

Once you've created the list environment, you create items for your list in the same way: `tags$li([content])`. You do not need to and should **NOT** use any header or paragraph tags with your list items; our Style Sheet will take care of the formatting. You can use emphasis/italics or strong/boldface on portions of the content as appropriate.

Here is an example of how you might create a list:

```
tags$ul(
  tags$li("This is my first item."),
  tags$li("A second item with a", tags$strong("boldface"), "word.")
)
```

which turns into

- This is my first item.
- A second item with a **boldface** word.

There is one exception to the environment that you need to be aware of: the Dashboard Header has a built-in listing environment and thus you will jump straight to `tags$li()` when in that section.

6.4.2 Headers

Heading tags provide a navigational structure to your app. Think of them as being the different levels of titles in a book. In fact, if you look at the Table of Contents to the left (provided you haven't hidden it), everything you see there is actually tied to a Header in this document.

No matter how much new coders want, Heading Tags are **NOT** for making text larger, boldface, or other text styling. Think about the headings as laying out a Table of Contents for your app, rather than containing content. Just like the Table of Contents for this Style Guide.

There is a specific ordering to the Header tags that is critical to ensure your App is accessible by screen readers.

1. `h1()` is for the Title of your App and should be ONCE at the top of the first page that appears when you load the app (i.e., the Overview).
2. `h2()` is for Tab/Page titles within the App. These would correspond with the tab links you place in the dashboard's left side panel.
3. `h3()` is for titling Sections within a Tab/Page of the App. These might title the portion of the page that is for a game board, questions, answers, and graphs/plots.
4. `h4()` is for a Subsection within a section. You might use this to distinguish different sets of controls in a Controls section.
5. `h5()` and `h6()` should be used sparingly. These might be used for different levels of a game. When you call these in your App, you just call them as listed here; i.e., `h1()`, `h2()`, etc.

Avoid skipping heading levels as this will get your App flagged for an Accessibility Violation. That is to say don't start at `h2` with no `h1` and don't go from `h1` to `h3` without an `h2`. Again, think of these as the layers of your table of contents or the outline of a paper; you wouldn't skip whole sections in either of those.

Here are few more things to NOT do with the heading tags:

- DO NOT USE headings to style text (We cannot stress this enough.)
- Do not wrap a header tag around a list element (e.g., `h3(tags$li("here is my list item"))`) nor the reverse (e.g., `tags$li(h3("here is my list item"))`)
- Do not mix header tags together in the same line or with the paragraph tag (e.g., 'h2("Introduction to", p("my game")))

For more information check out

- W3C's Tutorial
- Penn State's Headings and Subheadings Accessibility

6.4.3 Paragraphs

Your content should be enclosed in the paragraph tag, `p()`. Even if your content is relatively brief or a mathematical expression, the paragraph tag will ensure that the content is sized correctly and readable. This tag tells screen readers and browsers what your content actually is.

6.5 CSS

Cascading Style Sheets (CSS) will be the way to control the visual appearance of all elements of the app. To ensure that we have consistent we will make use of the BOAST CSS throughout. This is relatively new (Winter 2019/Spring 2020), so many of the older apps will need to have this added to their code. This allows us to centralize and dynamically update all apps at once.

There are a few key elements at play for CSS:

1. All apps need to have the appropriate reference to the CSS file (See Section 5.2.4).
   a. Please register any issues, bugs, enhancements, new styling to the Style Guide repository of GitHub.
2. Any App specific styling needs to as be in a CSS file and approved.
3. There should not be any styling code in the `app.R`, `ui.R`, or `server.R` files (a.k.a. in-line CSS). Styling code would look like this:
   - `tags$style(HTML(".js-irs-0 .irs-single, .js-irs-0 .irs-bar-edge, .js-irs-0 .irs-bar {background: orange}"))`
   - `style="color: #fff; background-color: #337ab7; border-color: #2e6da4"`
4. For objects that need to have a particular styling invoked, we will make use of the `class` attribute. These will invoked through the HTML tags or through the function calls.

```r
# HTML Tag Example 1
p(
  class = "hangingindent",
  "Attali, D. and Edwards, T. (2018). shinyalert: Easily create
  pretty popup messages (modals) in 'Shiny'. (v1.0). [R package].
  Available from https://CRAN.R-project.org/package=shinyalert"
)

# HTML Tag Example 2
div(class = "updated", "Last Update: 12/2/2019 by NJH.")

# Function Call Example
actionButton(
  inputId = paste0("grid-", row, "-", column),
  label = "?",
  color = "primary",
  style = "bordered",
  class = "grid-fill"
)
```

There is an exception to #3: setting up alignment for a div section. This would look like `div(style = "text-align: center"...)` This type of styling is allowed in the R files (generally in the UI portions only).

If you come across an app that has in-line CSS OR calls to a CSS file that isn't `boast.css`, please log an issue and mention/assign Neil. (Most common external CSS files are `style.css` and `Feature(s).css`.)

6.6 Metadata

As of boastUtils `v0.1.10`, your App will need the following metadata in a file named DESCRIPTION (no file extension). This data is used to inform Learning Record Stores (LRS) about your App as well as provide information to instructors using the Instructor App (unreleased).

DESCRIPTION

```
Title: Sample App - A Lengthy Title
ShortName: Sample App
Date: 2020-12-31
Authors@R: c(
    person("Lorem", "Ipsum", email = "lorem.ipsum@psu.edu", role = c("aut", "cre")),
    person("Ang", "Li", email = "ang.li@psu.edu", role = c("ctb"))
  )
Chapter: Sample Chapter
Description: This app is focused on the common types of xyz.
LearningObjectives: c(
    "The student will learn to understand Concept A in way z.",
    "The student will learn to understand Concept B as description y"
  )
DisplayMode: Normal
URL: https://psu-eberly.shinyapps.io/Sample_App
BugReports: https://github.com/EducationShinyAppTeam/Sample_App/issues
License: CC-BY-NC-SA-4.0
Tags: simulation
Type: Shiny
```

6.7 Using rLocker

Because these apps are being developed for educational purposes, it is beneficial to collect data (logfiles) based on learning objectives and outcomes. The rlocker package was created to aid in this process. It is important to know that the package itself does not perform feats of magic and will require some tuning to get right.

6.7.1 Installing

```
library(devtools)
devtools::install_github("rpc5102/rlocker")
```

View Documentation

6.7.2 Setup

Core configuration is included in the boastUtils package. Simply use the boastApp wrapper function instead of shinyApp and you're done! See Creating an App for more information on how to use boastApp.

6.7.3 Usage

The main purpose of this package is to help create meaningful- structured- xAPI data. xAPI (Experience API) can be thought of as the "Who did what, where did they do it, and when did they do it?" in your App. Statements are often structured as Actor, Verb, and Object that can also Result in something. For more on xAPI, check out What is the Experience API? or experiment with Statements in the xAPI Lab.

For example:

> **Bob** (Actor) **clicked** (Verb) **submit** (Object).

In assessments:

> **Neil** (Actor) **answered** (Verb) **Question 1** (Object) **correctly** with the answer **true** (Result).

This is a good way to think about it when beginning to write collection functions. Which brings us to our next part, writing collection functions.

Out of the box, `rlocker` does not provide any collection functions for apps, only creation and storage mechanisms. Why is this? Every app is different! You know your app the best and what interactions are possible. Before storing data in a Learning Record Store (LRS) like Learning Locker, it is important to transform it in a way that makes sense.

boastUtils

If using the boastApp wrapper for your project, your App will automatically generate and store a few of the more typical Statements. Please refrain from creating additional versions of the following statement descriptions:

Statement	Description
launched	User has started the app.
experienced	User has visited a tab (page) within the app.
exited *	User has left the app.

An App should only have one launched and exited event but can have multiple experiences per session. Therefore, you may use experienced in other places within your App.

* Requires boastUtils >= 0.1.5.

Sample generator function

```
.generateStatement <- function(
  session,
  verb = NA,
  object = NA,
  description = NA,
  interactionType = NA,
  response = NA,
  success = NA,
  completion = FALSE)
{
  statement <- rlocker::createStatement(list(
    verb = verb,
    object = list(
      id = paste0(getCurrentAddress(session), "#", object),
      name = paste0(APP_TITLE),
      description = paste0("Question ", activeQuestion, ": ", description),
      interactionType = interactionType
    ),
    result = list(
      success = success,
      response = response,
      completion = completion
    )
  ))
  return(rlocker::store(session, statement))
}
```

Sample event observer

```
observeEvent(session$input[[id]], {

  # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
  # BEGIN VALIDATION                                                          #
  #                                                                           #
  # For this example,                                                         #
  #   We will check if someone answered a question in our game correctly.    #
  #   Your code should be different here!                                     #
  #                                                                           #

  # The input element we're observing.                                        #
  object <- id

  # The user's action verb. Most likely going to be answered in this case.    #
  # Run rlocker::getVerbList() if you are unsure of the options.              #
  verb <- "answered"

  # The correct answer to the question.                                       #
  answer <- gameSet[id, "answer"]

  # The user's answer to the question.                                        #
  response <- input$ans

  # A description of the question or the full question itself (if short).     #
  description <- gameSet[id,]$question

  # The type of question it is.                                               #
  # Run rlocker::getInteractionTypes() if you are unsure of the options.      #
  interactionType <- ifelse(
    gameSet[id,]$format == "numeric", "numeric", "choice"
  )

  # Was the question answered successfully?                                   #
  success <- input$ans == answer

  # Did this event trigger the completion of your activity?                   #
  completion <- ifelse(.appState == "continue", FALSE, TRUE)

  #                                                                           #
  # END VALIDATION                                                            #
  # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

  .generateStatement(
    session,
    object = object,
    verb = verb,
    description = description,
    response = response,
    interactionType = interactionType,
    success = success,
    completion = completion
  )
}
```

Additional code can be found in the examples folder as well as the main README of the rLocker project; apps such as the Hypothesis_Testing_Game have already been outfitted with rlocker. The statement generator above will eventually make its way into `boastUtils` once enough feedback is collected. If you are ever confused about how it works, feel free to reach out to Bob (rpc5102).

Did you know?

You can run `help(package = "rlocker")` to view the help files for this package or press `F1` while typing a function name to see the documentation for that specific function.

Chapter 7

Coding Style

Now that you've gone through the Getting Started chapters, we can turn our attention to the first part of the Style Guide: Coding. There are many aspects that fall under the heading of Coding Style.

7.1 General Coding Style

Our general practice is to make use of the tidyverse style guide. However, we do have some important departures and additional practices you need to follow:

- Leave comments
- Use informative names
- Format your code
- Be explicit

7.1.1 Leave Comments

At bare minimum, use a comment to break your code into sections. This helps you and others conceptualize the code into more manageable chunks. Your comments can provide others with potential keywords to search for when looking at your code later on.

For particularly complex sections, use comments to summarize what you're trying to do. This can help you and others pick up the coding thread for what you are trying to do for troubleshooting, debugging, and future improvements.

7.1.2 Informative Names

Use informative names for variables and functions in your code. Use names that give another person a sense of what that variable represents (nouns) or what the function is supposed to do (action verbs). For example,

- `scoreMatrix` is a matrix that holds a set of scores
- `checkGame` is a function that checks the state of the current game

Use camelCase for variable names and functions. The first word begins with a lowercase letter and additional words start with Capital letters with no spaces or underscores (_) between them. (This is a departure from the tidyverse style.)

Avoid using the variable names from code that you're making use of from another app or script. For example, don't use `waitTimes` to hold your data on the number of correct answers a user has given. It is also good practice to avoid re-using function names that appear in other libraries that are currently loaded to avoid namespace collision.

7.1.3 Format Your Code

See also Section 5.3

Use indentation spacing to help make your code readable. RStudio has a built-in tool that can help with this.

1. Select all of the code you want to reformat.
   - To select all code in a file, use Command-A (Mac) or Control-A (Windows).
2. Press Command-Shift-A (Mac), Control-Shift-A (Windows), or click on the Code menu and select Reformat Code.

NOTE: this tool is imperfect and can result in left parenthesis or curly braces moving up a line to where you might have an end-of-line comment, resulting in errors.

Additionally, you can make use of the `styler` and `lintr` packages to help you perform formatting checks and to quickly reformat code.

7.1.4 Be Explicit

One of the best practices a coder can do is to be explicit. And no, we don't mean that type of explicit. This links back to Informative Naming but goes a step beyond. `R` is a functional programming language. One of the important aspects of this is that functions in `R` obey the same rules as mathematical functions, especially multivariate functions.

One implicit fact about functions that most students don't realize is that the order of a function's arguments are a matter of convention and are actually arbitrary. While we might define f as $f(x, y) = x^2 + 3y$, we could call $f(y = 2, x = 1)$ and get the same output as $f(1, 2)$ when we use the convention set up in the definition. This issue is exacerbated with functions in `R`.

Therefore, when you pass values to the arguments of a function in R, you should be explicit and include the argument name in your code. For example,

```
actionButton(
  inputId = "submit",
  label = "Submit",
  color = "primary",
  style = "bordered",
  class = "btn-ttt"
)
```

There are a few exceptions to this:

- Formula Arguments (e.g., `response ~ factor1 + block`)
- Data Vectors/Frames (e.g., `dataFrame$response`)
- Functions from `shiny`, `shinydashboard`, `shinyBS`, etc.
  - E.g., you can call `dashboardHeader` rather than `shinydashboard::dashboardHeader`

General Advice:

- If you are calling a function from a package that is unique to your App or not commonly used, list the package in the function call.
- If you are using a common/central function (i.e., those from `shiny`, `ggplot2`), you can omit the package in the function call.

Remember the following: the more proactive you are from the get go in commenting and organizing your code, the easier time you will have for debugging and improving your code down the road.

7.2 Additional Coding Style

In addition, we use the following coding practices.

7.2.1 Minimize Package Usage

Make sure that you absolutely have to use a particular package before you do. Check to see if what you're trying to do can be done in a package you're already using or in base R.

This is not to say that you can't make use of a new package. This guideline's purpose is to streamline the various packages that get used in BOAST and to use the most of the packages that we are using. However, there are times when we just need to use a new package. Please try to use packages that are housed on CRAN and are preferably under active development. Searching GitHub for the package name can help you decide whether the package is active.

To help you avoid name masking (i.e., Section 5.1.4), ensure that you are actually using a package, and to help future readers follow your code, explicitly call

packages with their functions. For example, use dplyr::filter([arguments]) instead of just filter([arguments]).

You can also run a funchir::stale_package_check on your app.R, ui.R, and sever.R files to see which packages you're loading but might not actually use in your code. These stale packages may then be deleted out from your library call.

```r
# Check working directory
getwd()
# Does the output match the folder path to where you app lives?
# If not, then you need to set that. For example,
setwd("~/Documents/GitHub/shiny-apps/Sample_App")

# Run a stale package check on app.R, ui.R, and server.R
funchir::stale_package_check("app.R")
```

Be aware that while stale_package_check is useful, it doesn't always catch everything. For example, when we ran it on the sample app, we were told that there were no exported functions from ggplot2 or tippy. However, we know that there are functions from both of those. If you have a giant list of libraries to check, there might some more misses.

Once you're sure that a package isn't being used, remove the library call for that package from your code.

If you are dealing with a ui.R/server.R pair, make sure that the appropriate packages are loaded in the right file. For example, ggplot2 only needs to be loaded in the server.R file.

7.2.2 Minimize External Files

Try to minimize the number and size of external files you're attaching to your App. If you're working on an existing app, remove any external files that are no longer necessary.

Wherever possible try to place external files into the www directory/folder that is at the same level as your app.R or ui.R/server.R files.

If there are any external files (e.g., *.csv, *.txt, *.dat, *.jpg) that are not being used, delete them from the repository.

7.2.3 Plot Caching

There are cases when your app will include a plot that falls into any of the following categories:

1) a plot that all users will see (i.e., static data set),
2) is a computationally intense plot (i.e., lots of data and/or layers),

3) might be a dynamic graph that the user explores and needs to move back and forth between various states.

Each of these cases represents a place where the performance of your app can suffer. To improve performance, you should consider using the `renderCachePlot` function rather than `renderPlot`. This function will store a copy of each plot on the sever and provide that stored copy to new instances of the app. This cuts down on server demands and speeds up your app. For the third category, this allows the users to move more quickly to a previously examined state (i.e., low to no hang time).

7.2.4 Styling Text

The styling of text (including font size, font weight (i.e., boldface), font family, color, etc.) is something that we began standardizing in Fall 2019. This is centrally managed by Neil and Bob to ensure 1) consistency across apps, 2) to cut down on the extraneous coding you need to do (this way you can focus on the apps), and 3) ensures that our apps are accessible and mobile friendly. To this end:

The styling of text will be controlled using an **external** CSS (Cascading Style Sheet) file. You will implement this depending on what approach you use:

- If you're using the `boastApp` function from `boastUtils` (**recommended for all new apps**), you don't need to do anything as the appropriate calls will automatically be done for you.
- If you are using the `ui.R` and `server.R` format (i.e., apps from prior years), place the following code in the `tags$head( )` portion at the top of the `dashboardBody` section of the `ui.R` file :

```
dashboardPage(
  skin = "blue",
  dashboardHeader(
    title = "title",
    titleWidth = 300,
    # [omitted code]
  ),
  # [omitted code],
  dashboardBody(
    tags$head(
      tags$link(rel = "stylesheet", type = "text/css",
        href = "https://educationshinyappteam.github.io/Style_Guide/theme/boast.css"),
      tabItems(
       # [omitted code]
      )  )  )  )
```

There times when you might need some additional text styling that is not already defined. In these cases, **you'll need to talk to Neil or Bob to determine**

what is the best course of action. This could entail an addition to the central CSS file or the inclusion of an additional CSS file unique to your App.

The central CSS file should only be altered by Neil or Bob and covers many aspects. However, this is a living file meaning that we will add to/alter the file as needed to improve the whole set of BOAST apps.

See also Section 5.5.

7.3 Organizing Your Code

There is a fixed order in which you should write your code. This will depend on if you are using a singular `app.R` file or the pair `ui.r` and `server.r`.

7.3.1 Using `app.R`

The order for your code will be as follows:

1. Packages to be loaded
2. App Metadata (see below)
3. Any additional source files and custom functions
4. UI definition (see Chapter 8)
5. Server definition
6. `boastApp` call

7.3.2 Using `ui.R` and `server.R`

The order for your code in the `ui.R` file:

1. Packages to be loaded
2. App Metadata (see below)
3. Any additional source files and custom functions
4. UI definition (see Chapter 8)

The order for your code in the `server.R` file:

1. Packages to be loaded
2. Any additional source files and custom functions
3. Server definition

7.4 HTML

Our Shiny apps live on the internet and are thus going to be wrapped in HTML even though we are writing them in `R`. When you run a Shiny app, `R` and the `shiny` packages actually convert all of the `R` code into an HTML document which is served up to the user. While you will not have to directly write HTML for your apps, you should become aware of HTML tags that you will need to

deal with when coding/writing your App. Further, you need to learn how to use these tags correctly as failure to do so becomes an **Accessibility Issue**.

HTML Tags should not be used without some basic understanding of what that tag is for. Using the tags without such an understanding can not only lead to problems with your code running properly or looking like what you intended, but can interfere with how accessible your app is to a wider audience. Here are the most common HTML tags that you'll encounter when making a Shiny App.

7.4.1 Lists

There are two aspects to lists: the items in the list and the environment those items live in.

The two list environments are Ordered Lists and Unordered Lists.

- The Ordered List environment is for when you require that a user works through the items in a particular order. You call this environment in your App by using `tags$ol( [your list] )`. These lists are sequentially numbered/lettered.
- The Unordered List environment is for when you want to give the user a list where they can jump around between the items however they wish. You call this environment in your App by using `tags$ul( [your list] )`. These lists will appear with bullets.

Once you've created the list environment, you create items for your list in the same way: `tags$li([content])`. You do not need to and should **NOT** use any header or paragraph tags with your list items; our Style Sheet will take care of the formatting. You can use emphasis/italics or strong/boldface on portions of the content as appropriate.

Here is an example of how you might create a list:

```
tags$ul(
  tags$li("This is my first item."),
  tags$li("A second item with a", tags$strong("boldface"), "word.")
)
```

which turns into

- This is my first item.
- A second item with a **boldface** word.

There is one exception to the environment that you need to be aware of: the Dashboard Header has a built-in listing environment and thus you will jump straight to `tags$li()` when in that section.

7.4.2 Headers

Heading tags provide a navigational structure to your app. Think of them as being the different levels of titles in a book. In fact, if you look at the Table of Contents to the left (provided you haven't hidden it), everything you see there is actually tied to a Header in this document.

No matter how much new coders want, Heading Tags are **NOT** for making text larger, boldface, or other text styling. Think about the headings as laying out a Table of Contents for your app, rather than containing content. Just like the Table of Contents for this Style Guide.

There is a specific ordering to the Header tags that is critical to ensure your App is accessible by screen readers.

1. `h1()` is for the Title of your App and should be ONCE at the top of the first page that appears when you load the app (i.e., the Overview).
2. `h2()` is for Tab/Page titles within the App. These would correspond with the tab links you place in the dashboard's left side panel.
3. `h3()` is for titling Sections within a Tab/Page of the App. These might title the portion of the page that is for a game board, questions, answers, and graphs/plots.
4. `h4()` is for a Subsection within a section. You might use this to distinguish different sets of controls in a Controls section.
5. `h5()` and `h6()` should be used sparingly. These might be used for different levels of a game. When you call these in your App, you just call them as listed here; i.e., `h1()`, `h2()`, etc.

Avoid skipping heading levels as this will get your App flagged for an Accessibility Violation. That is to say don't start at `h2` with no `h1` and don't go from `h1` to `h3` without an `h2`. Again, think of these as the layers of your table of contents or the outline of a paper; you wouldn't skip whole sections in either of those.

Here are few more things to NOT do with the heading tags:

- DO NOT USE headings to style text (We cannot stress this enough.)
- Do not wrap a header tag around a list element (e.g., `h3(tags$li("here is my list item"))`) nor the reverse (e.g., `tags$li(h3("here is my list item"))`)
- Do not mix header tags together in the same line or with the paragraph tag (e.g., 'h2("Introduction to", p("my game")))

For more information check out

- W3C's Tutorial
- Penn State's Headings and Subheadings Accessibility

7.4.3 Paragraphs

Your content should be enclosed in the paragraph tag, `p()`. Even if your content is relatively brief or a mathematical expression, the paragraph tag will ensure that the content is sized correctly and readable. This tag tells screen readers and browsers what your content actually is.

7.5 CSS

Cascading Style Sheets (CSS) will be the way to control the visual appearance of all elements of the app. To ensure that we have consistent we will make use of the BOAST CSS throughout. This is relatively new (Winter 2019/Spring 2020), so many of the older apps will need to have this added to their code. This allows us to centralize and dynamically update all apps at once.

There are a few key elements at play for CSS:

1. All apps need to have the appropriate reference to the CSS file (See Section 5.2.4).
   a. Please register any issues, bugs, enhancements, new styling to the Style Guide repository of GitHub.
2. Any App specific styling needs to as be in a CSS file and approved.
3. There should not be any styling code in the `app.R`, `ui.R`, or `server.R` files (a.k.a. in-line CSS). Styling code would look like this:
   - `tags$style(HTML(".js-irs-0 .irs-single, .js-irs-0 .irs-bar-edge, .js-irs-0 .irs-bar {background: orange}"))`
   - `style="color: #fff; background-color: #337ab7; border-color: #2e6da4"`
4. For objects that need to have a particular styling invoked, we will make use of the `class` attribute. These will invoked through the HTML tags or through the function calls.

```r
# HTML Tag Example 1
p(
  class = "hangingindent",
  "Attali, D. and Edwards, T. (2018). shinyalert: Easily create
  pretty popup messages (modals) in 'Shiny'. (v1.0). [R package].
  Available from https://CRAN.R-project.org/package=shinyalert"
)

# HTML Tag Example 2
div(class = "updated", "Last Update: 12/2/2019 by NJH.")

# Function Call Example
actionButton(
  inputId = paste0("grid-", row, "-", column),
  label = "?",
  color = "primary",
  style = "bordered",
  class = "grid-fill"
)
```

There is an exception to #3: setting up alignment for a div section. This would look like `div(style = "text-align: center"...)` This type of styling is allowed in the R files (generally in the UI portions only).

If you come across an app that has in-line CSS OR calls to a CSS file that isn't `boast.css`, please log an issue and mention/assign Neil. (Most common external CSS files are `style.css` and `Feature(s).css`.)

7.6 Metadata

As of boastUtils `v0.1.10`, your App will need the following metadata in a file named DESCRIPTION (no file extension). This data is used to inform Learning Record Stores (LRS) about your App as well as provide information to instructors using the Instructor App (unreleased).

DESCRIPTION

```
Title: Sample App - A Lengthy Title
ShortName: Sample App
Date: 2020-12-31
Authors@R: c(
    person("Lorem", "Ipsum", email = "lorem.ipsum@psu.edu", role = c("aut", "cre")),
    person("Ang", "Li", email = "ang.li@psu.edu", role = c("ctb"))
  )
Chapter: Sample Chapter
Description: This app is focused on the common types of xyz.
LearningObjectives: c(
    "The student will learn to understand Concept A in way z.",
    "The student will learn to understand Concept B as description y"
  )
DisplayMode: Normal
URL: https://psu-eberly.shinyapps.io/Sample_App
BugReports: https://github.com/EducationShinyAppTeam/Sample_App/issues
License: CC-BY-NC-SA-4.0
Tags: simulation
Type: Shiny
```

7.7 Using rLocker

Because these apps are being developed for educational purposes, it is beneficial to collect data (logfiles) based on learning objectives and outcomes. The rlocker package was created to aid in this process. It is important to know that the package itself does not perform feats of magic and will require some tuning to get right.

7.7.1 Installing

```r
library(devtools)
devtools::install_github("rpc5102/rlocker")
```

View Documentation

7.7.2 Setup

Core configuration is included in the boastUtils package. Simply use the boastApp wrapper function instead of shinyApp and you're done! See Creating an App for more information on how to use boastApp.

7.7.3 Usage

The main purpose of this package is to help create meaningful- structured- xAPI data. xAPI (Experience API) can be thought of as the "Who did what, where did they do it, and when did they do it?" in your App. Statements are often structured as Actor, Verb, and Object that can also Result in something. For more on xAPI, check out What is the Experience API? or experiment with Statements in the xAPI Lab.

For example:

> **Bob** (Actor) **clicked** (Verb) **submit** (Object).

In assessments:

> **Neil** (Actor) **answered** (Verb) **Question 1** (Object) **correctly** with the answer **true** (Result).

This is a good way to think about it when beginning to write collection functions. Which brings us to our next part, writing collection functions.

Out of the box, `rlocker` does not provide any collection functions for apps, only creation and storage mechanisms. Why is this? Every app is different! You know your app the best and what interactions are possible. Before storing data in a Learning Record Store (LRS) like Learning Locker, it is important to transform it in a way that makes sense.

boastUtils

If using the boastApp wrapper for your project, your App will automatically generate and store a few of the more typical Statements. Please refrain from creating additional versions of the following statement descriptions:

Statement	Description
launched	User has started the app.
experienced	User has visited a tab (page) within the app.
exited *	User has left the app.

An App should only have one launched and exited event but can have multiple experiences per session. Therefore, you may use experienced in other places within your App.

* Requires boastUtils >= 0.1.5.

Sample generator function

```
.generateStatement <- function(
  session,
  verb = NA,
  object = NA,
  description = NA,
  interactionType = NA,
  response = NA,
  success = NA,
  completion = FALSE)
{
  statement <- rlocker::createStatement(list(
    verb = verb,
    object = list(
      id = paste0(getCurrentAddress(session), "#", object),
      name = paste0(APP_TITLE),
      description = paste0("Question ", activeQuestion, ": ", description),
      interactionType = interactionType
    ),
    result = list(
      success = success,
      response = response,
      completion = completion
    )
  ))
  return(rlocker::store(session, statement))
}
```

Sample event observer

```
observeEvent(session$input[[id]], {

  # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
  # BEGIN VALIDATION                                                          #
  #                                                                           #
  # For this example,                                                         #
  #   We will check if someone answered a question in our game correctly.     #
  #   Your code should be different here!                                     #
  #                                                                           #

  # The input element we're observing.                                        #
  object <- id

  # The user's action verb. Most likely going to be answered in this case.    #
  # Run rlocker::getVerbList() if you are unsure of the options.              #
  verb <- "answered"

  # The correct answer to the question.                                       #
  answer <- gameSet[id, "answer"]

  # The user's answer to the question.                                        #
  response <- input$ans

  # A description of the question or the full question itself (if short).     #
  description <- gameSet[id,]$question

  # The type of question it is.                                               #
  # Run rlocker::getInteractionTypes() if you are unsure of the options.      #
  interactionType <- ifelse(
    gameSet[id,]$format == "numeric", "numeric", "choice"
  )

  # Was the question answered successfully?                                   #
  success <- input$ans == answer

  # Did this event trigger the completion of your activity?                   #
  completion <- ifelse(.appState == "continue", FALSE, TRUE)

  #                                                                           #
  # END VALIDATION                                                            #
  # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

  .generateStatement(
    session,
    object = object,
    verb = verb,
    description = description,
    response = response,
    interactionType = interactionType,
    success = success,
    completion = completion
  )
}
```

Additional code can be found in the examples folder as well as the main README of the rLocker project; apps such as the Hypothesis_Testing_Game have already been outfitted with rlocker. The statement generator above will eventually make its way into `boastUtils` once enough feedback is collected. If you are ever confused about how it works, feel free to reach out to Bob (rpc5102).

Did you know?

You can run `help(package = "rlocker")` to view the help files for this package or press `F1` while typing a function name to see the documentation for that specific function.

Part IV

Style Guide-App Layout

Chapter 8

App Layout

When considering the Visual Appearance of your App, there are two major areas of consideration: the Layout and what we refer to as Design Styling. Each of these will be handled in turn. Keep in mind that these two aspects are interrelated and have significant cross-over.

In this chapter, we will be focusing on the layout aspect of how your App looks. That is to say, we're going to talk about the standards/guidelines that cut across all of our apps so that they look like they belong together.

A good number of the elements of the layout will have a direct consequence on your coding. For instance, in the section on Organizing Your Code, you saw that the UI definition needed to come before the Server definition. This section defines additional organizational structure to your code as it pertains to the layout of your App.

As a reminder, one of the most important benefits of using the `boastApp` function from the `boastUtils` package is that is that certain aspects of Visual Appearance will be automatically handled for you. However, you still need to do adhere to this Style Guide.

8.1 Dashboard

All apps will make use of a Dashboard structure. This divides the visual appearance of each App into three main areas.

- Across the top of the App will be the Header
- Along the left side of the App will be the navigation list (the Sidebar) where the various Tabs (pages) of your App will be listed
- The last area is the Body; this is where all content will appear

Several of the older apps will have outdated UI calls including, but not limited to: shinyUI and navbarPage. These functions should no longer be used; apps that use them need to be updated to become compliant with this Style Guide.

8.1.1 Creating the Dashboard

Creating the overarching Dashboard layout in your App is actually quit easy:

```r
# Required package
library(shinydashboard)

# For app.R files
# [code omitted]
ui <- list(
  dashboardPage(
    skin = "blue",
    # [code omitted]
  )
)

# For ui.R files
dashboardPage(
  skin = "blue",
  # [code omitted]
)
```

The dashboardPage function will wrap around the rest of your UI element, thereby establishing the overarching structure. Three of this function's arguments (header, sidebar, and body) will be covered in following sections. We do not need to worry about the title argument as this will be controlled by the Dashboard Header.

The one argument that you need to explicitly set for the dashboardPage is the skin argument. This argument sets the overall color theme for your App. While this is something that is more an aspect of Design Style, your only opportunity to set this value is here with the Layout.

8.1.2 Color in the User Interface

Within BOAST, we use color themes to help provide consistency for the elements of each app and to denote different chapters. Part of the standardization process of this Style Guide seeks to bring the many fractured color themes together into a cohesive, centrally managed set. This helps reduce the programming burden on the students, who should focus on the R side of the programming, not the CSS side.

All aspects of color in the User Interface should be controlled through the CSS file(s). This includes all of the following:

- Dashboard coloring (Header, Sidepanel, Body)
- Text coloring
- Coloring of Controls (including buttons, sliders, and other input fields)

By using CSS, especially through `boastApp`, you'll be able to ensure that there is consistent coloring throughout your App.

8.1.2.1 Implementing a Color Theme

To activate a color theme is a simple process, especially if you are following this Style Guide and using the `boastUtils` package. (If you are in an App using ui.R and server.R, make sure that the boast.css call is in the ui.R file. See Section 5.2.4.)

In your App's code, go to where you first call the function `dashboardPage`. Then, as the first argument you'll type `skin = "[theme]"` before moving on the next argument, `dashboardHeader`.

You will replace `[theme]` with one of the following: `blue`, `green`, `purple`, `yellow`, `red` or `black`. The choice will be determined by the color assigned to that chapter. This is all you have to do.

If you are unsure what color to put, use `blue` as the default.

8.1.2.2 The Themes

There are six color themes that we've currently made. The names of the themes are a general indication of coloring, with one exception. The `black` theme is not black but rather an aqua/teal set. The themes are typically three colors (four for `blue`) and based upon the Penn State Palettes. Non-Penn State colors will be denoted with asterisks.

All of the themes have been checked against 8 different forms of color blindness.

8.1.2.2.1 Blue The Blue Palette is our central palette and should be used by default. The Blue Palette looks like the following:

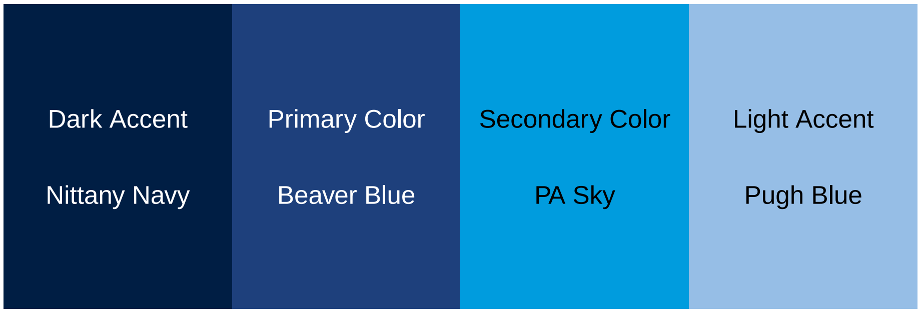


Figure 8.1: The Blue Palette

Here is what the Blue Palette looks like in action:

8.1.2.2.2 Green

The Green Palette looks like the following:

Here is what the Green Palette looks like in action:

8.1.2.2.3 Purple

The Purple Palette looks like the following:

Here is what the Purple Palette looks like in action:

8.1.2.2.4 Black

The "Black" Palette is not pegged to the color black, but rather teal/aqua colors. However, to call the theme in the Shiny dashboard, the user must use the value `black` for the the `skin` argument. Here's what the "Black" Palette looks like:

Here is what the "Black" Palette looks like in action:

8.1.2.2.5 Yellow

The Yellow Palette is still under consideration. The current set looks like the following:

Here is what the Yellow Palette looks like in action:

8.1.2.2.6 Red

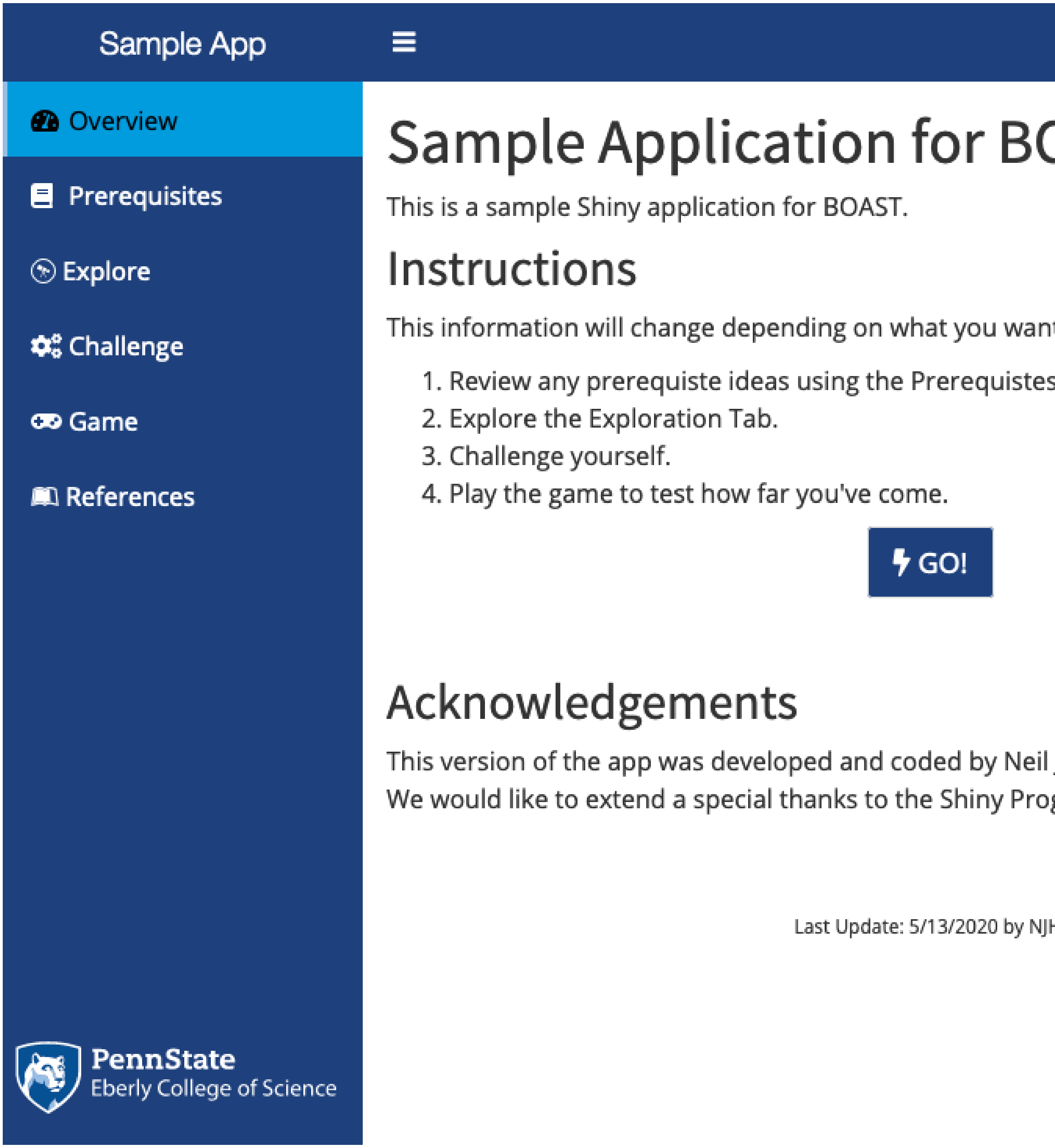
The Red Palette is still under construction. Here's the current set:

Here is what the Red Palette looks like in action:

8.1.3 Current Chapter Color Assignments

Here are the current (05/27/2020) color theme assignments for chapters:

- Chapter 1: Data Gathering RED
- Chapter 2: Data Description YELLOW
- Chapter 3: Basic Probability BLUE
- Chapter 4: Statistical Inference PURPLE
- Chapter 5: Probability BLUE
- Chapter 6: Regression "BLACK"
- Chapter 7: ANOVA "BLACK"
- Chapter 8: Time Series PURPLE
- Chapter 9: Sampling RED
- Chapter 10: Categorical Data YELLOW
- Chapter 11: Data Science GREEN
- Chapter 12: Stochastic Processes BLUE
- Chapter 13: Biology GREEN



Sample Application for BC

This is a sample Shiny application for BOAST.

Instructions

This information will change depending on what you want

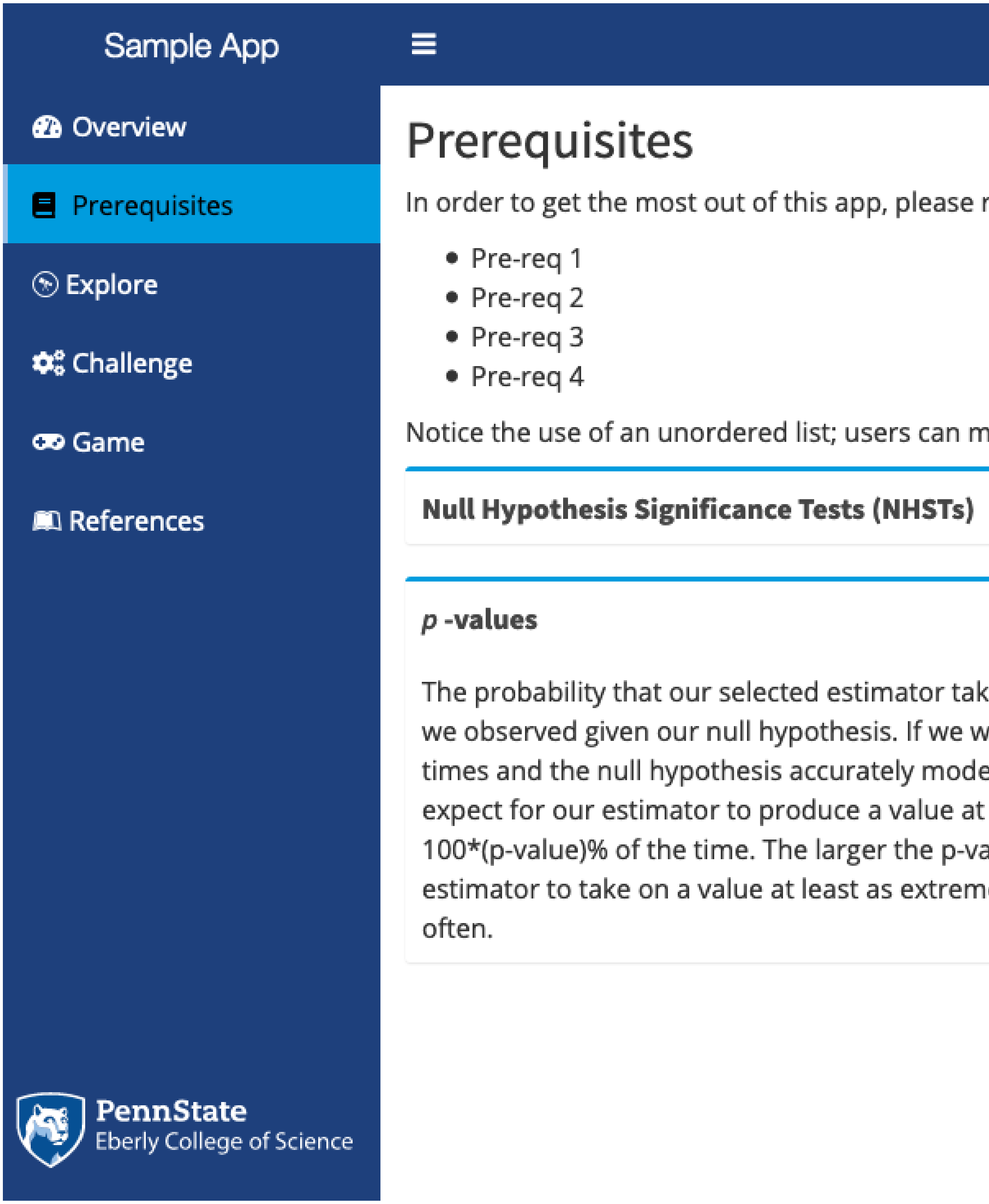
1. Review any prerequiste ideas using the Prerequistes
2. Explore the Exploration Tab.
3. Challenge yourself.
4. Play the game to test how far you've come.

Acknowledgements

This version of the app was developed and coded by Neil
We would like to extend a special thanks to the Shiny Prog

Last Update: 5/13/2020 by NJH

Figure 8.2: Overview Page Using the Blue Palette



Prerequisites

In order to get the most out of this app, please r

- Pre-req 1
- Pre-req 2
- Pre-req 3
- Pre-req 4

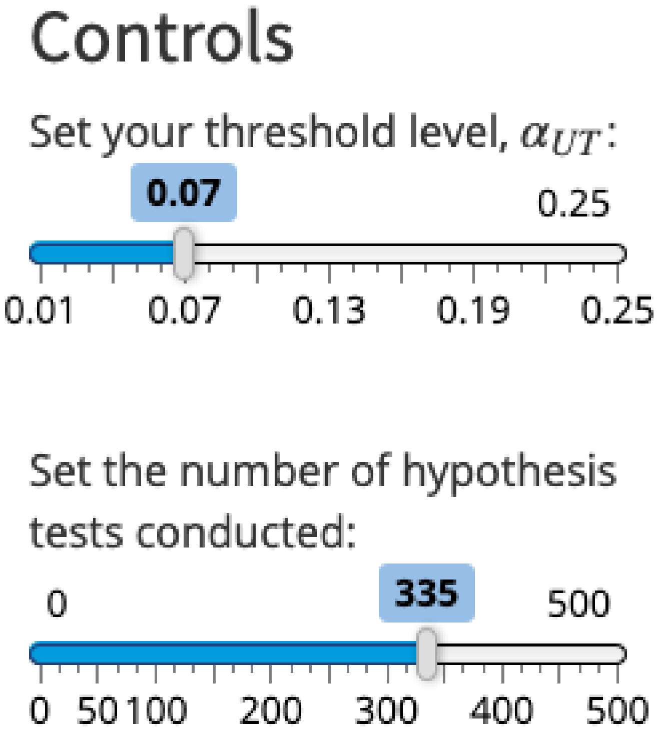
Notice the use of an unordered list; users can m

Null Hypothesis Significance Tests (NHSTs)

***p* -values**

The probability that our selected estimator tak
we observed given our null hypothesis. If we w
times and the null hypothesis accurately mode
expect for our estimator to produce a value at
100*(p-value)% of the time. The larger the p-va
estimator to take on a value at least as extrem
often.

Figure 8.3: Collapsible Boxes Using the Blue Palette



Controls

Set your threshold level, α_{UT}:

0.07

0.25

Set the number of hypothesis tests conducted:

0

335

500

Figure 8.4: Sliders Using the Blue Palette

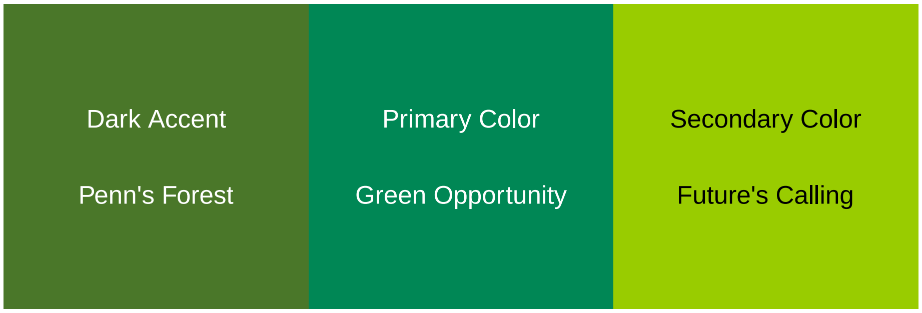
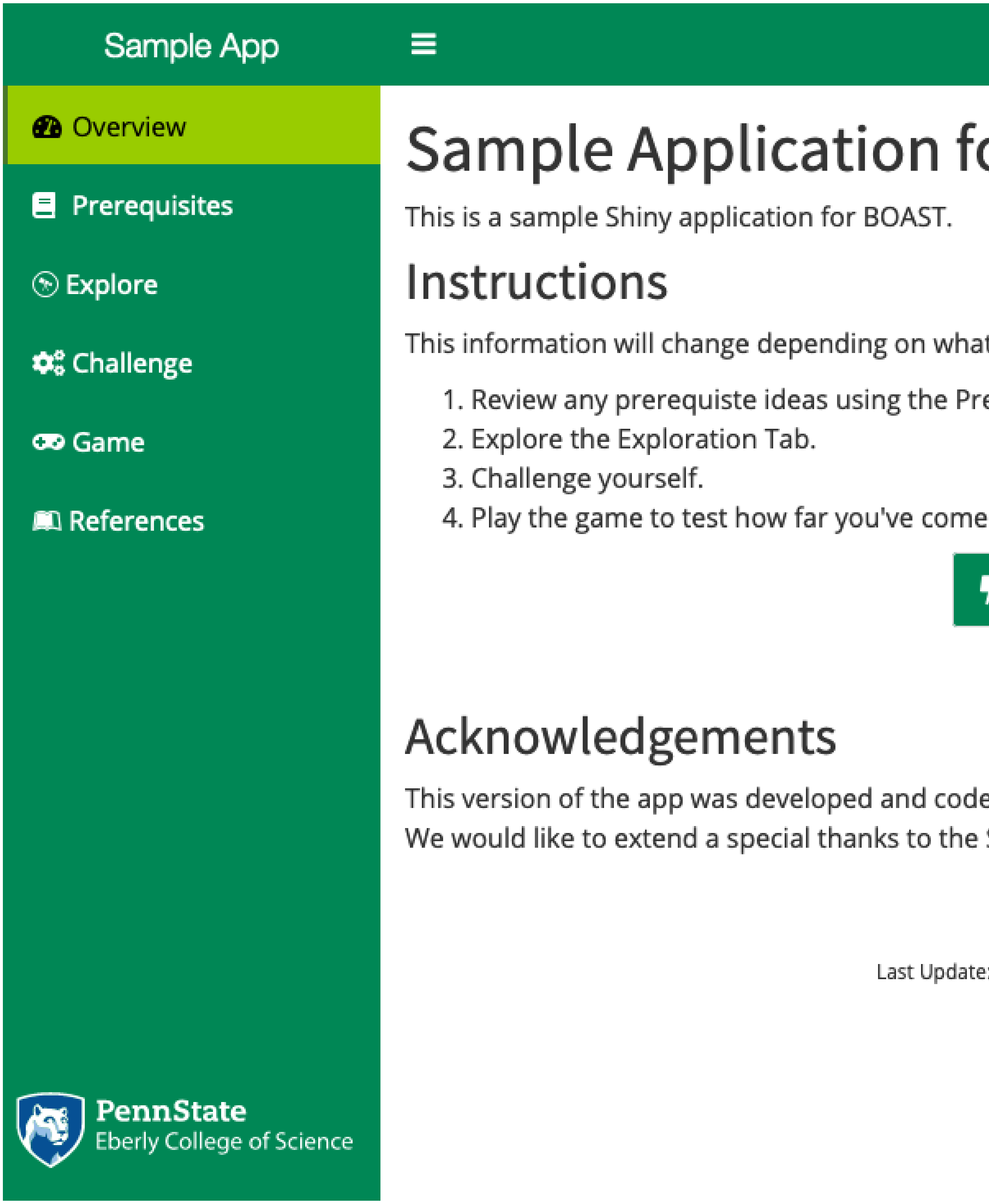


Figure 8.5: The Green Palette



Sample Application fo

This is a sample Shiny application for BOAST.

Instructions

This information will change depending on what

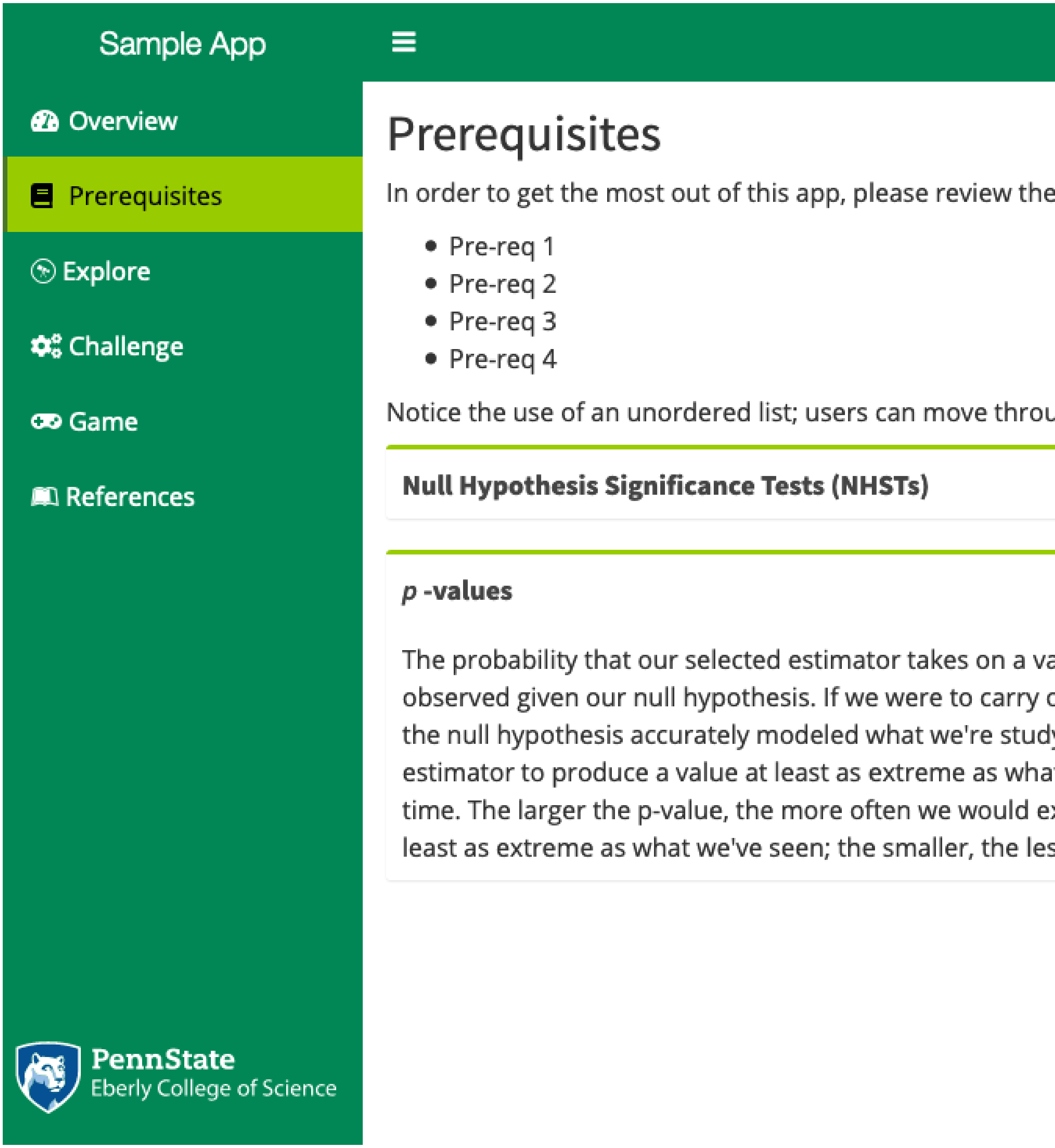
1. Review any prerequiste ideas using the Pre
2. Explore the Exploration Tab.
3. Challenge yourself.
4. Play the game to test how far you've come

Acknowledgements

This version of the app was developed and code
We would like to extend a special thanks to the S

Last Update:

Figure 8.6: Overview Page Using the Green Palette



Prerequisites

In order to get the most out of this app, please review the

- Pre-req 1
- Pre-req 2
- Pre-req 3
- Pre-req 4

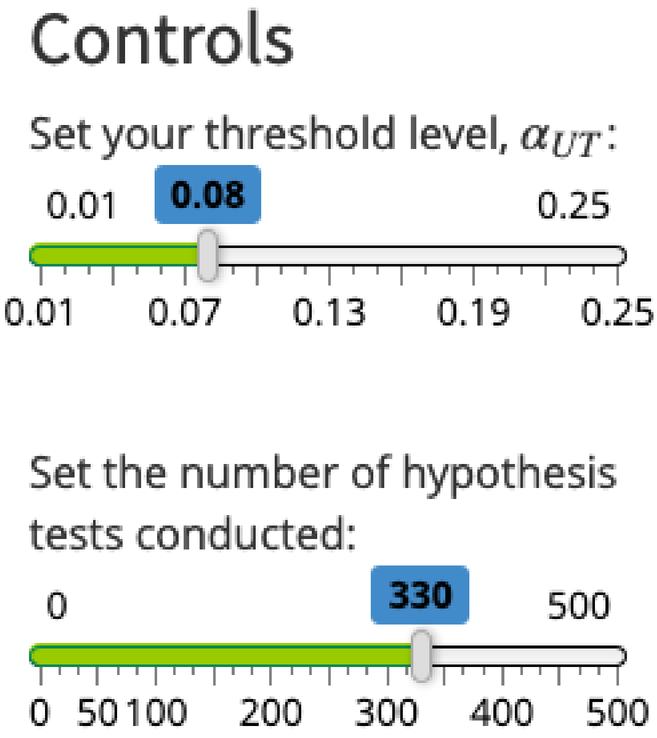
Notice the use of an unordered list; users can move throu

Null Hypothesis Significance Tests (NHSTs)

p **-values**

The probability that our selected estimator takes on a va observed given our null hypothesis. If we were to carry c the null hypothesis accurately modeled what we're stud estimator to produce a value at least as extreme as wha time. The larger the p-value, the more often we would e: least as extreme as what we've seen; the smaller, the les

Figure 8.7: Collapsible Boxes Using the Green Palette



Controls

Set your threshold level, α_{UT}:

0.08

Set the number of hypothesis tests conducted:

330

Figure 8.8: Sliders Using the Green Palette

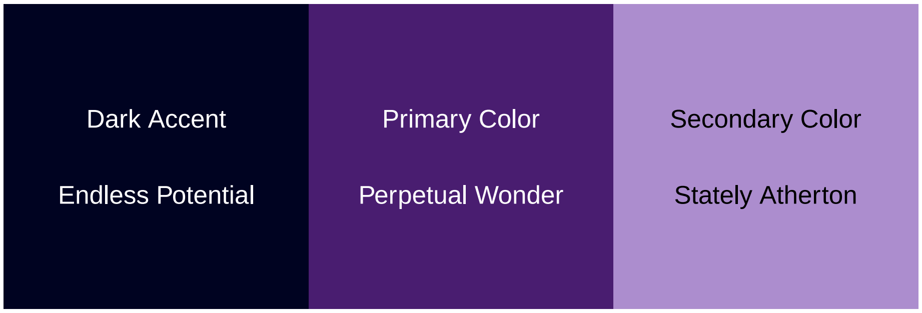
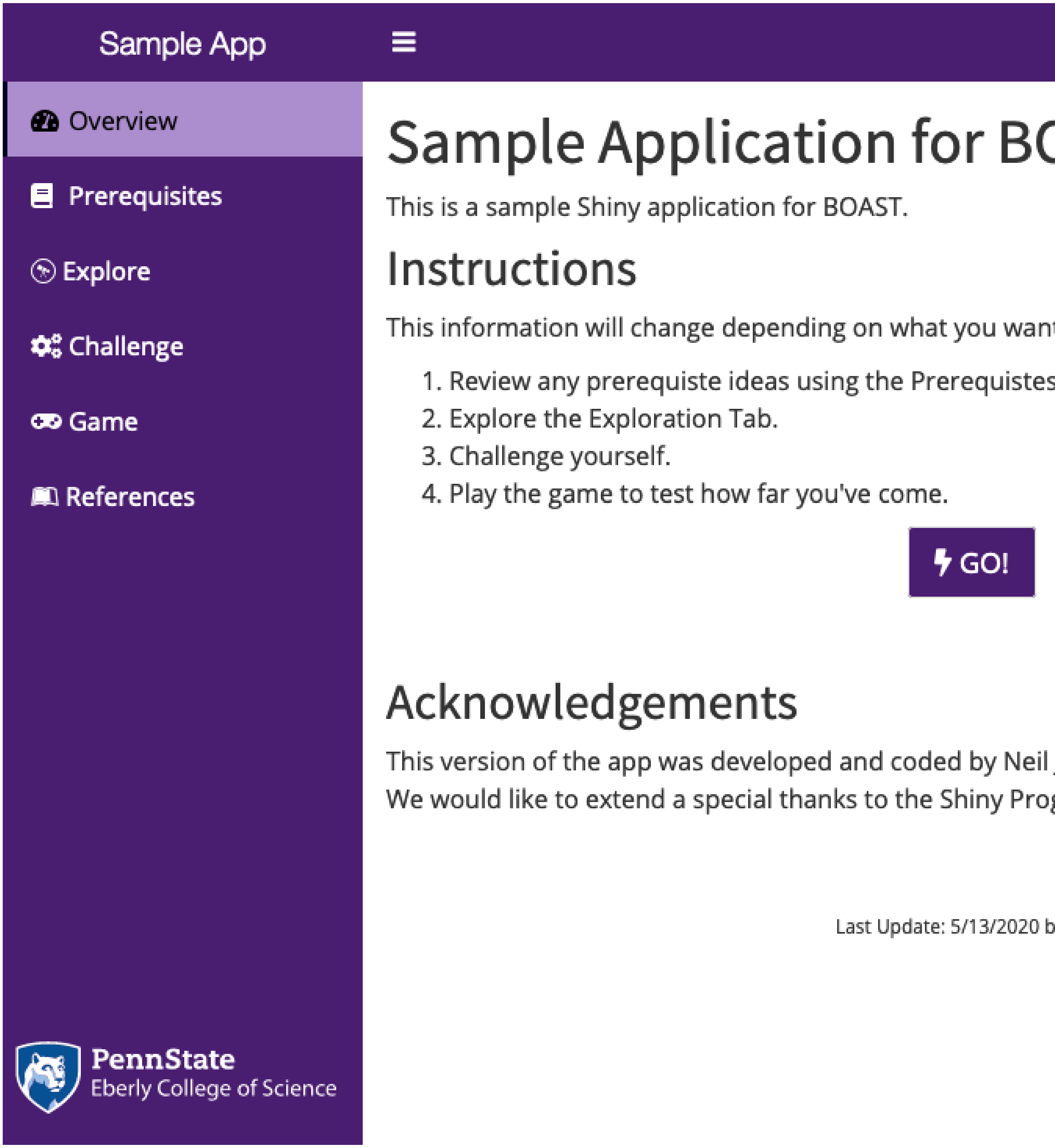


Figure 8.9: The Purple Palette



Sample Application for BO

This is a sample Shiny application for BOAST.

Instructions

This information will change depending on what you want

1. Review any prerequiste ideas using the Prerequistes
2. Explore the Exploration Tab.
3. Challenge yourself.
4. Play the game to test how far you've come.

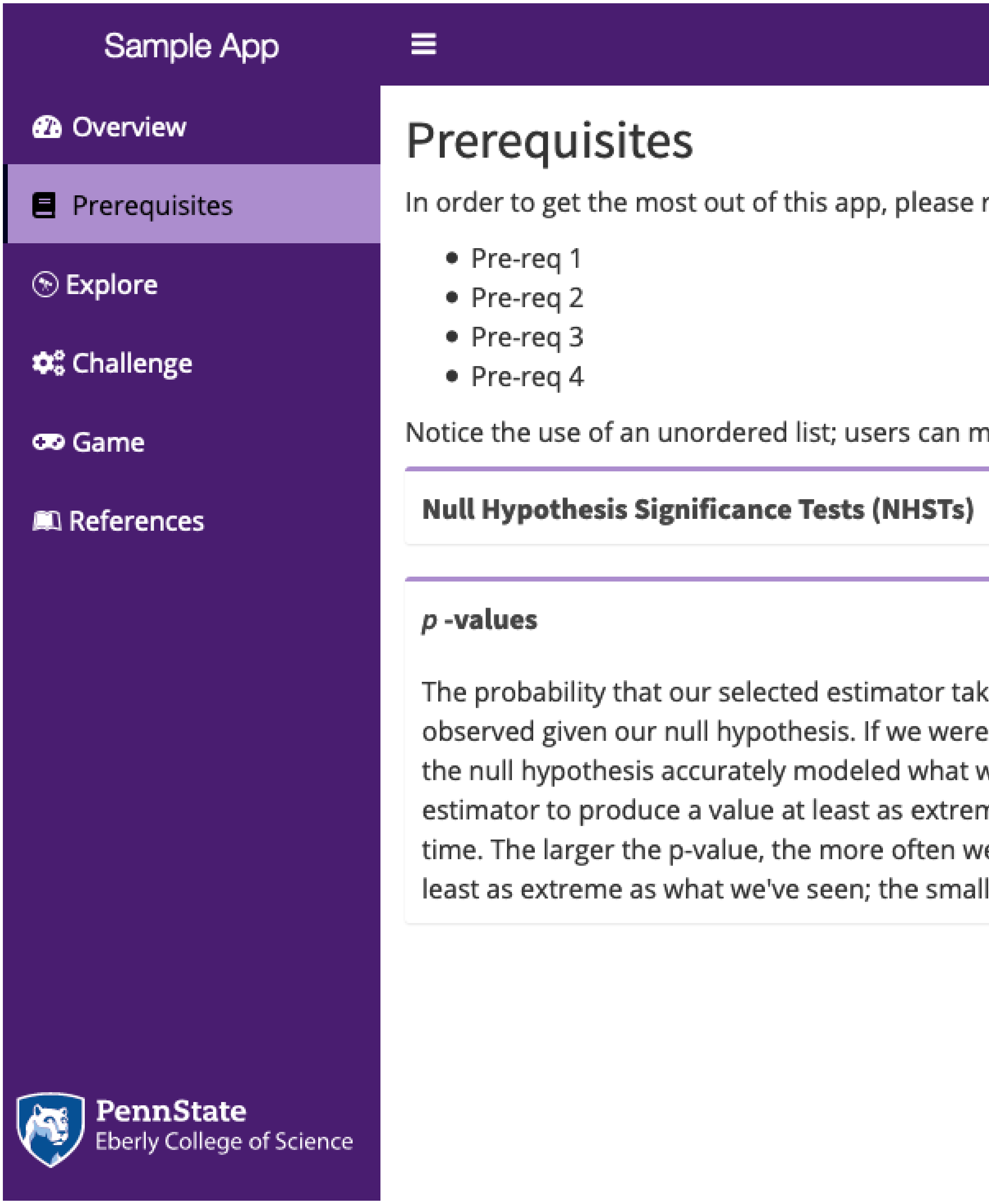
Acknowledgements

This version of the app was developed and coded by Neil

We would like to extend a special thanks to the Shiny Prog

Last Update: 5/13/2020 b

Figure 8.10: Overview Page Using the Purple Palette



Prerequisites

In order to get the most out of this app, please r

- Pre-req 1
- Pre-req 2
- Pre-req 3
- Pre-req 4

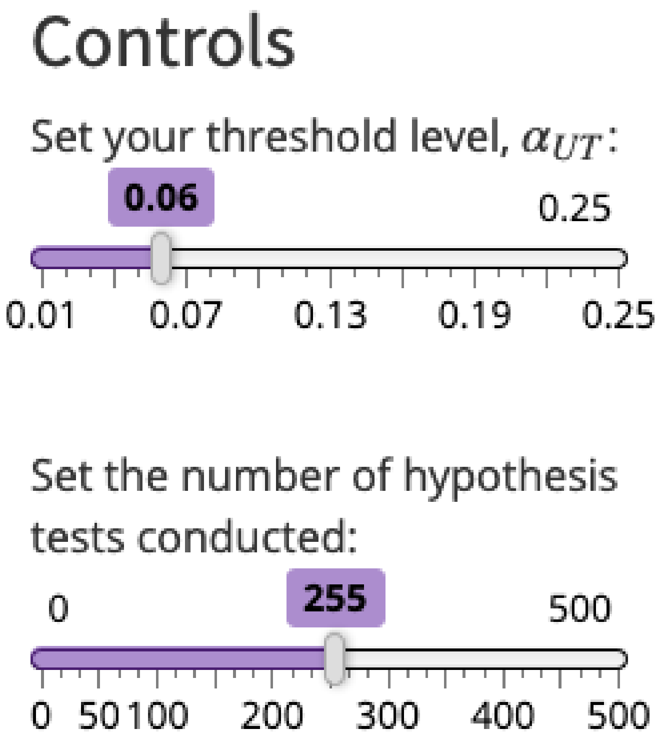
Notice the use of an unordered list; users can m

Null Hypothesis Significance Tests (NHSTs)

***p* -values**

The probability that our selected estimator tak
observed given our null hypothesis. If we were
the null hypothesis accurately modeled what w
estimator to produce a value at least as extrem
time. The larger the p-value, the more often we
least as extreme as what we've seen; the small

Figure 8.11: Collapsible Boxes Using the Purple Palette



Controls

Set your threshold level, α_{UT}:

0.06

Set the number of hypothesis tests conducted:

255

Figure 8.12: Sliders Using the Purple Palette

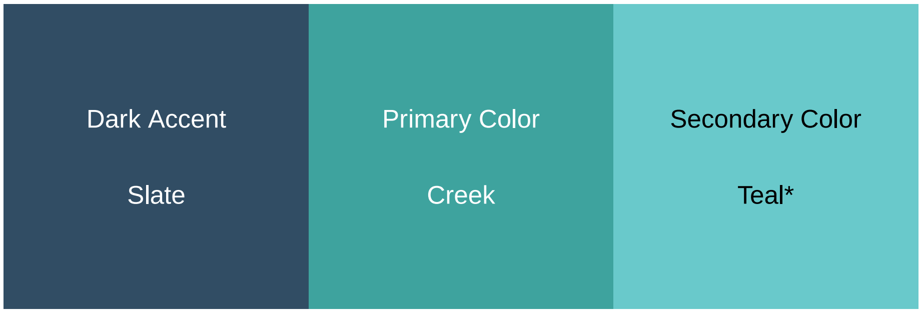
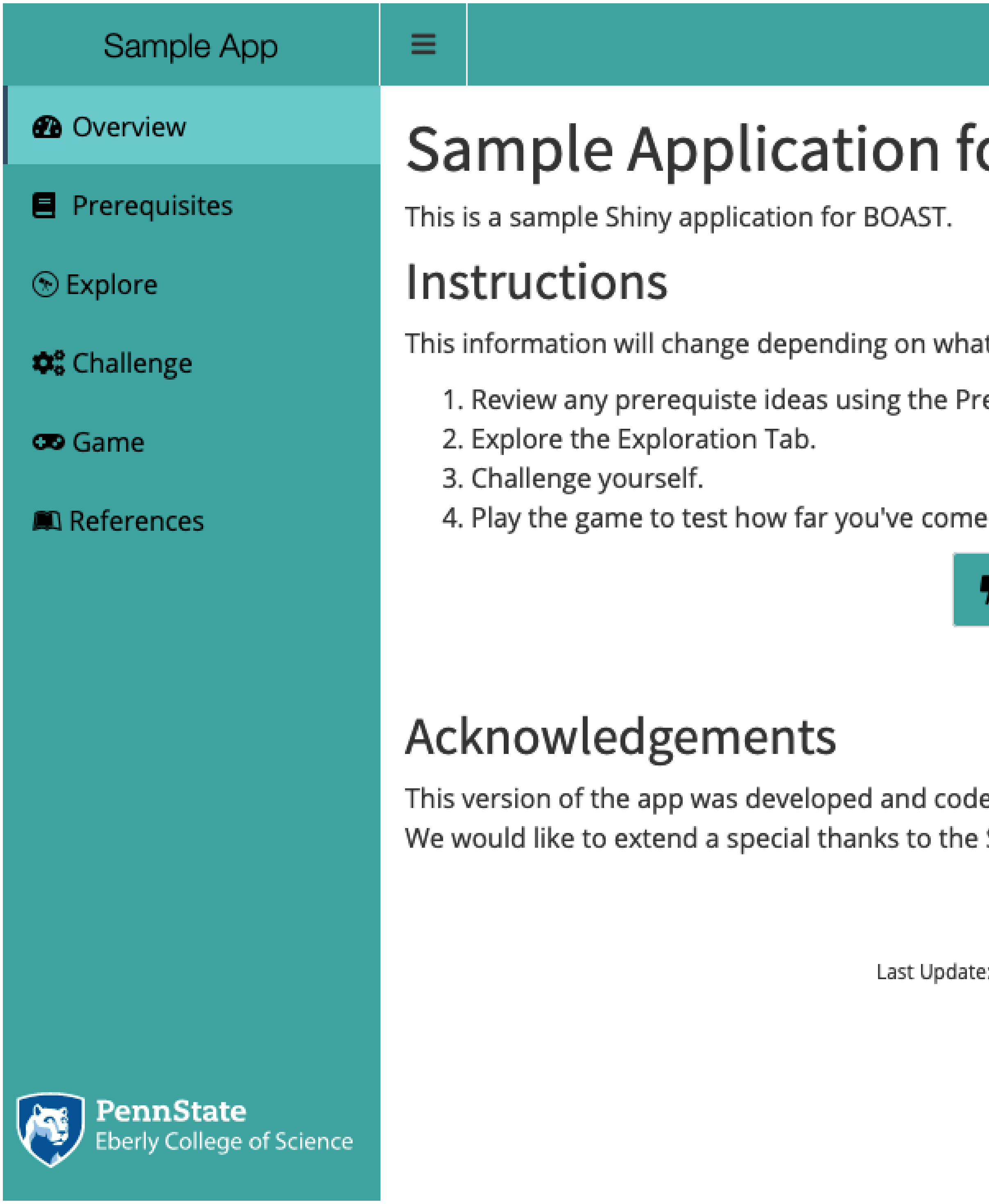


Figure 8.13: The 'Black' Palette



Sample Application fo

This is a sample Shiny application for BOAST.

Instructions

This information will change depending on what

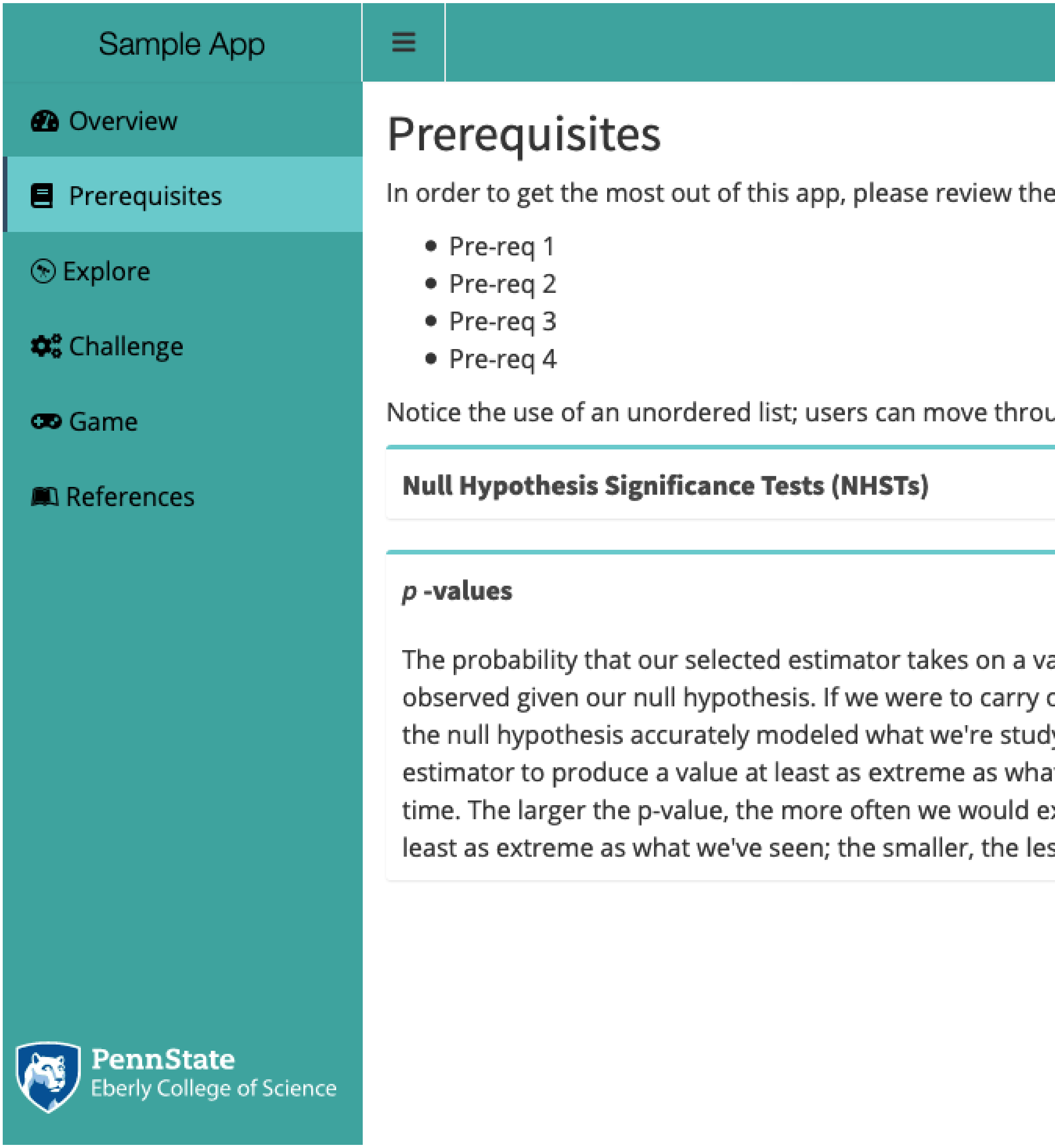
1. Review any prerequiste ideas using the Pre
2. Explore the Exploration Tab.
3. Challenge yourself.
4. Play the game to test how far you've come

Acknowledgements

This version of the app was developed and code
We would like to extend a special thanks to the

Last Update

Figure 8.14: Overview Page Using the 'Black' Palette



Prerequisites

In order to get the most out of this app, please review the

- Pre-req 1
- Pre-req 2
- Pre-req 3
- Pre-req 4

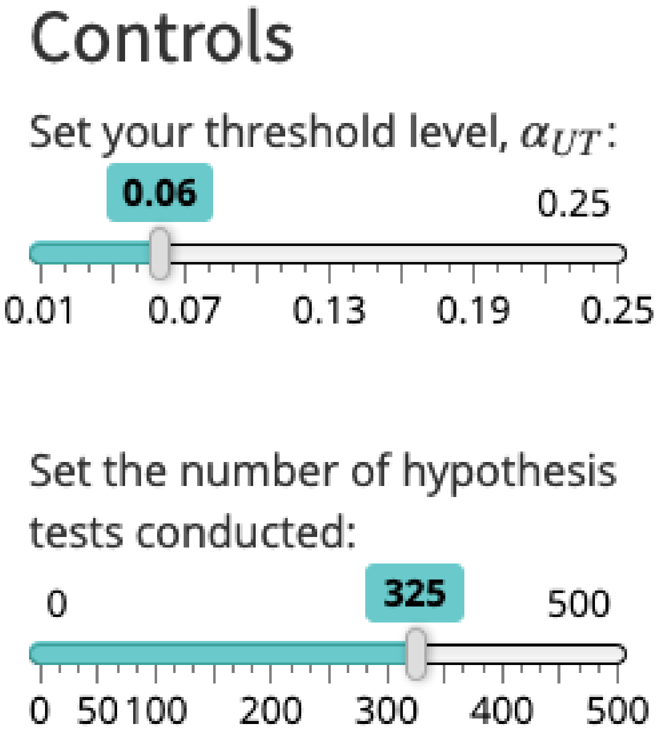
Notice the use of an unordered list; users can move throu

Null Hypothesis Significance Tests (NHSTs)

p **-values**

The probability that our selected estimator takes on a va
observed given our null hypothesis. If we were to carry o
the null hypothesis accurately modeled what we're study
estimator to produce a value at least as extreme as wha
time. The larger the p-value, the more often we would ex
least as extreme as what we've seen; the smaller, the les

Figure 8.15: Collapsible Boxes Using the 'Black' Palette



Controls

Set your threshold level, α_{UT}:

0.06

Set the number of hypothesis tests conducted:

325

Figure 8.16: Sliders Using the 'Black' Palette

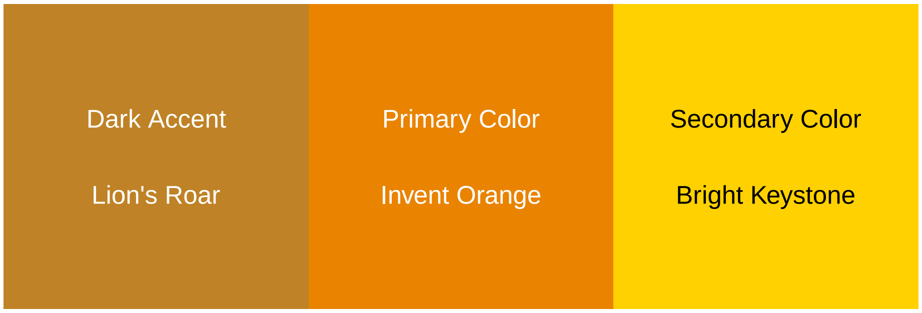
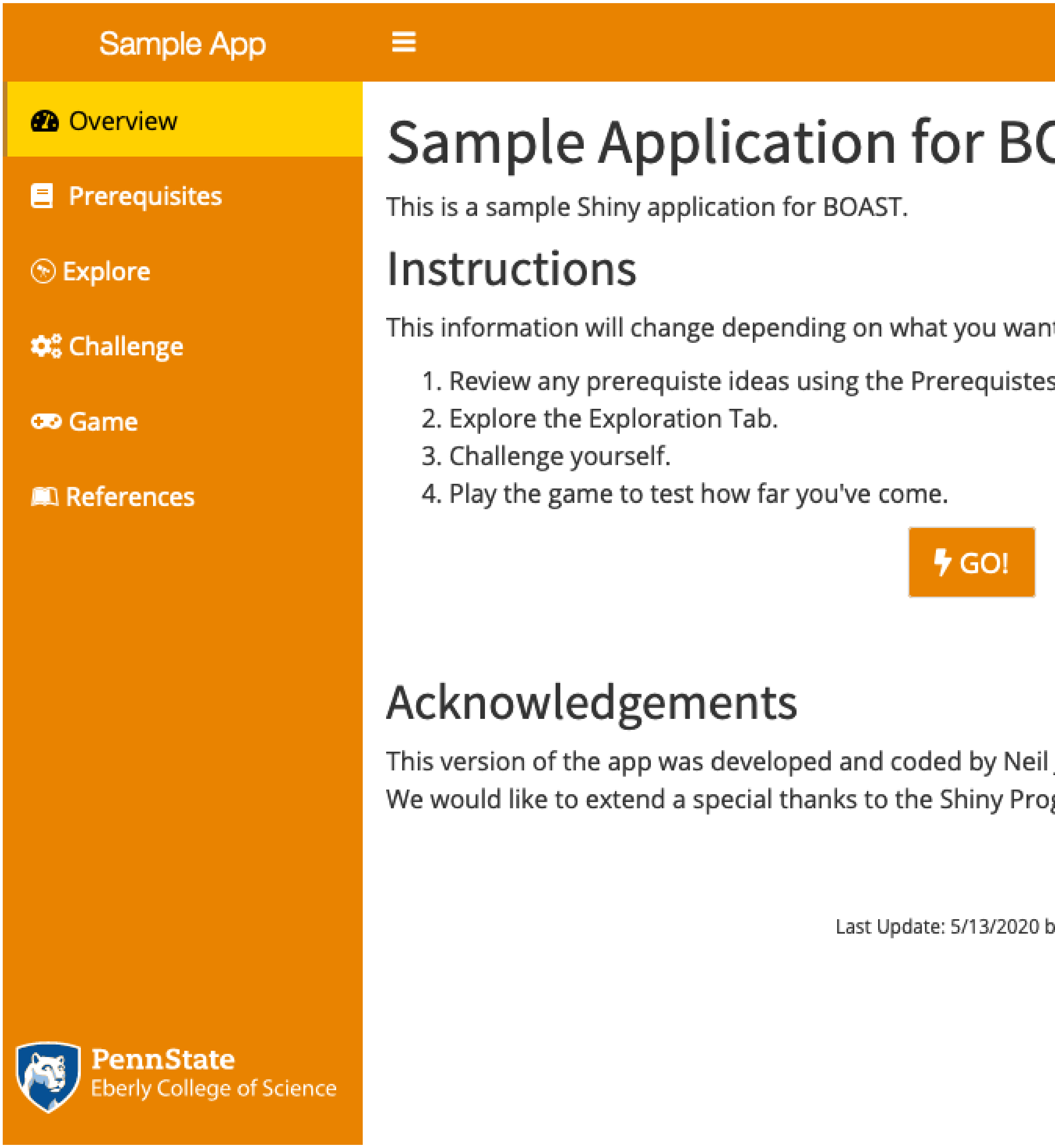


Figure 8.17: The Yellow Palette



Sample Application for BO

This is a sample Shiny application for BOAST.

Instructions

This information will change depending on what you want

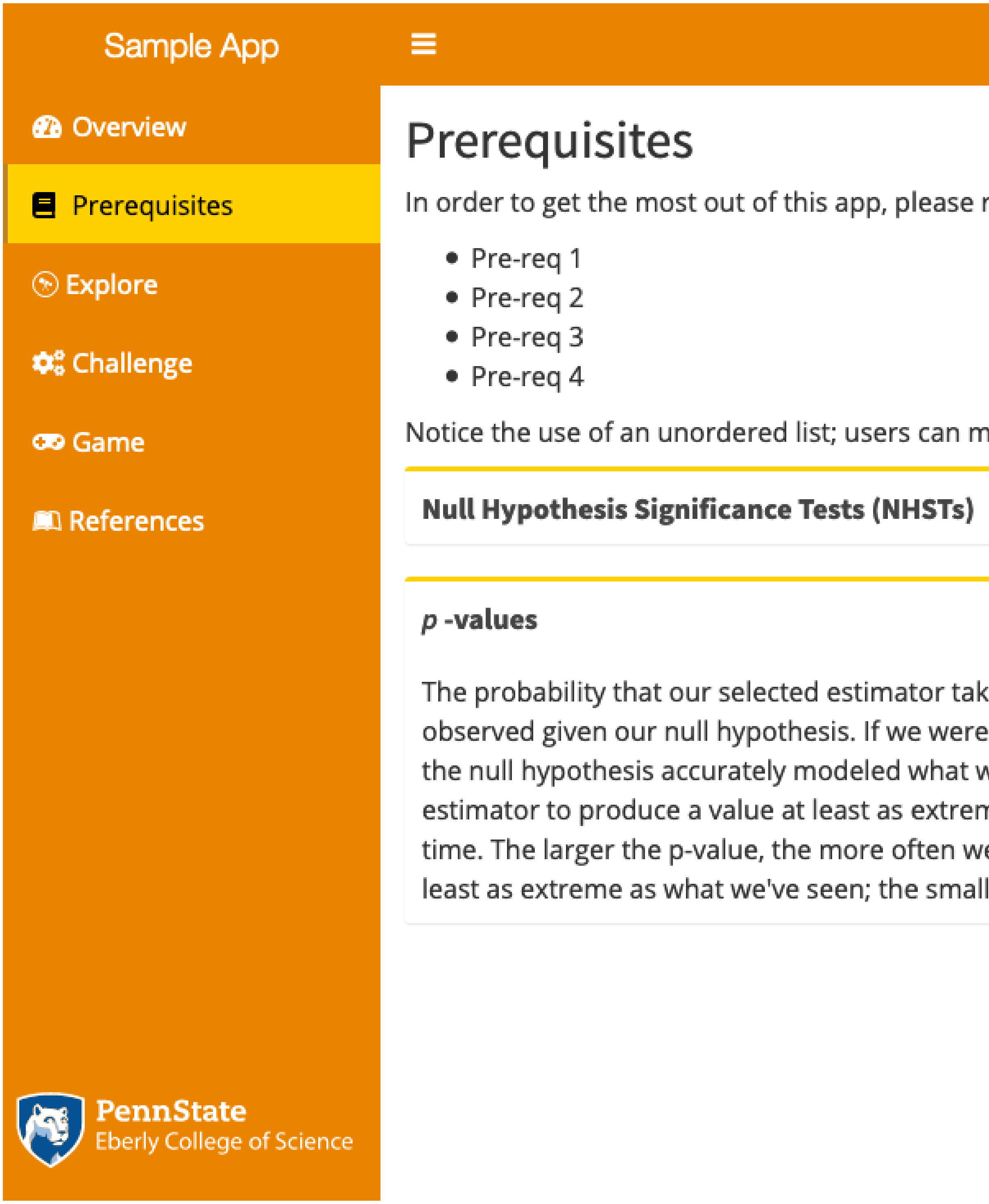
1. Review any prerequiste ideas using the Prerequistes
2. Explore the Exploration Tab.
3. Challenge yourself.
4. Play the game to test how far you've come.

Acknowledgements

This version of the app was developed and coded by Neil
We would like to extend a special thanks to the Shiny Prog

Last Update: 5/13/2020 b

Figure 8.18: Overview Page Using the Yellow Palette



Prerequisites

In order to get the most out of this app, please r

- Pre-req 1
- Pre-req 2
- Pre-req 3
- Pre-req 4

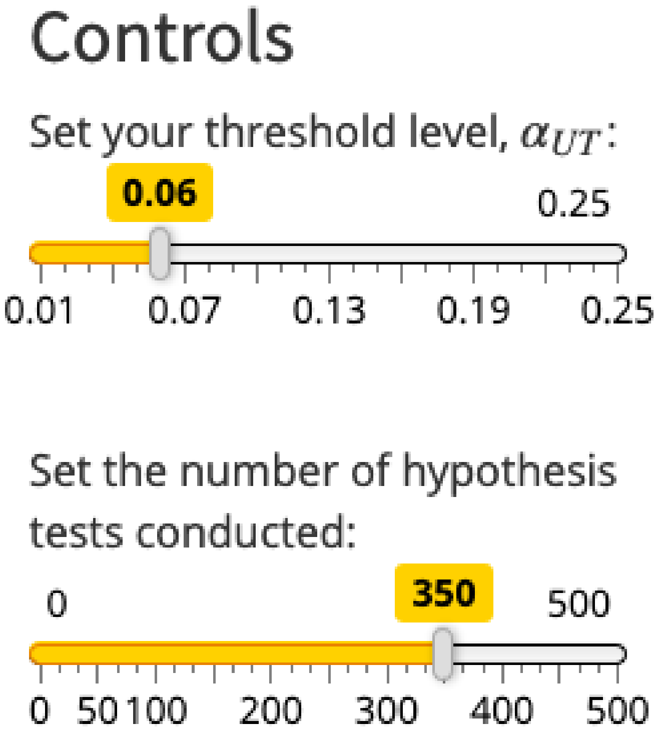
Notice the use of an unordered list; users can m

Null Hypothesis Significance Tests (NHSTs)

***p* -values**

The probability that our selected estimator tak
observed given our null hypothesis. If we were
the null hypothesis accurately modeled what w
estimator to produce a value at least as extrem
time. The larger the p-value, the more often we
least as extreme as what we've seen; the small

Figure 8.19: Collapsible Boxes Using the Yellow Palette



Controls

Set your threshold level, α_{UT}:

0.06

Set the number of hypothesis tests conducted:

350

Figure 8.20: Sliders Using the Yellow Palette

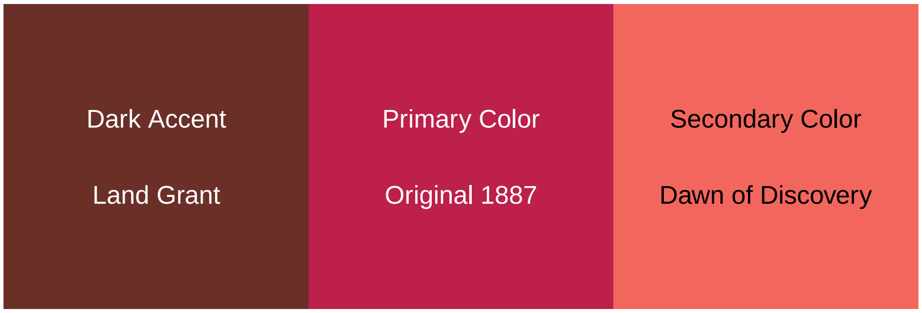
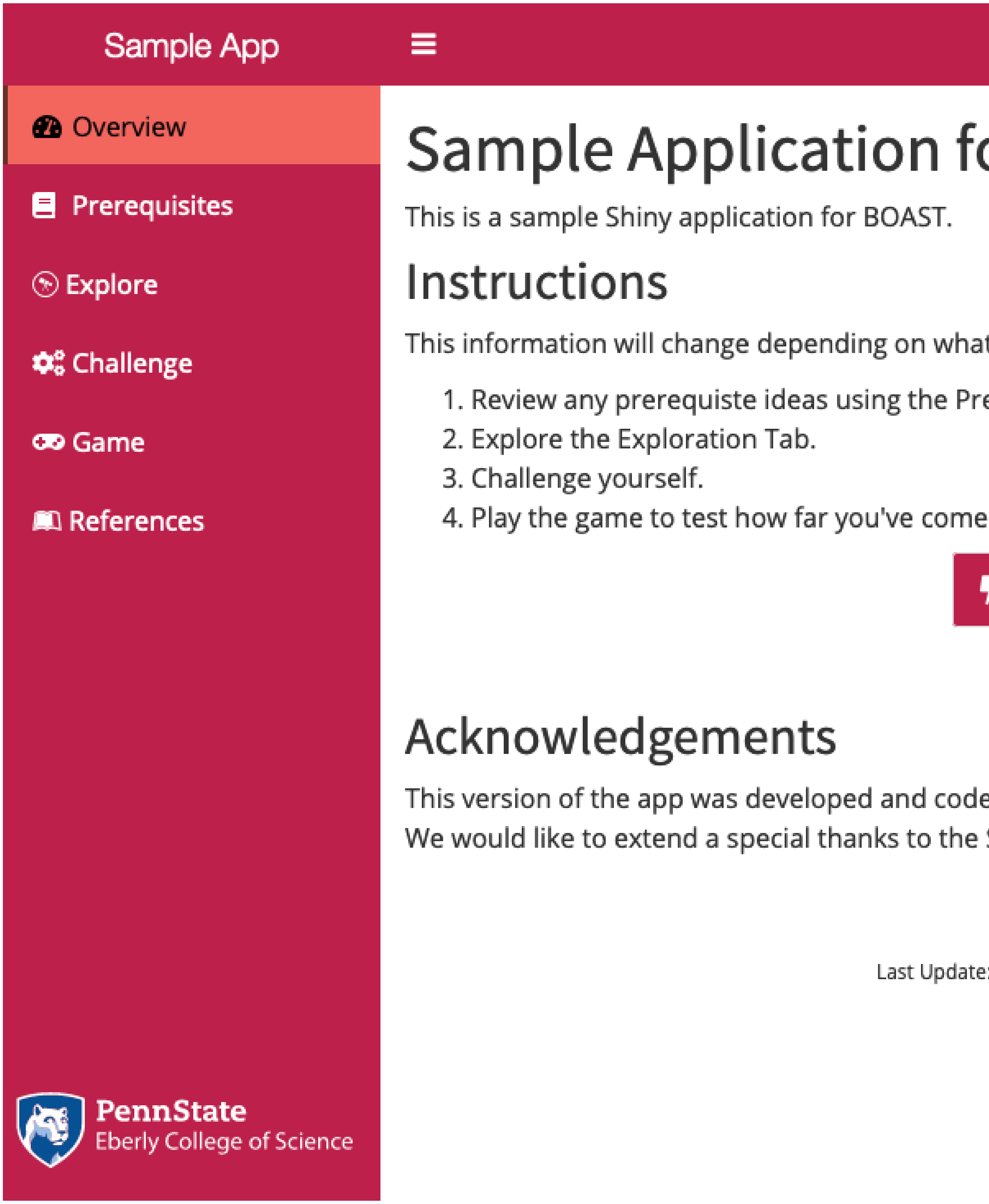


Figure 8.21: The Red Palette



Sample Application fo

This is a sample Shiny application for BOAST.

Instructions

This information will change depending on what

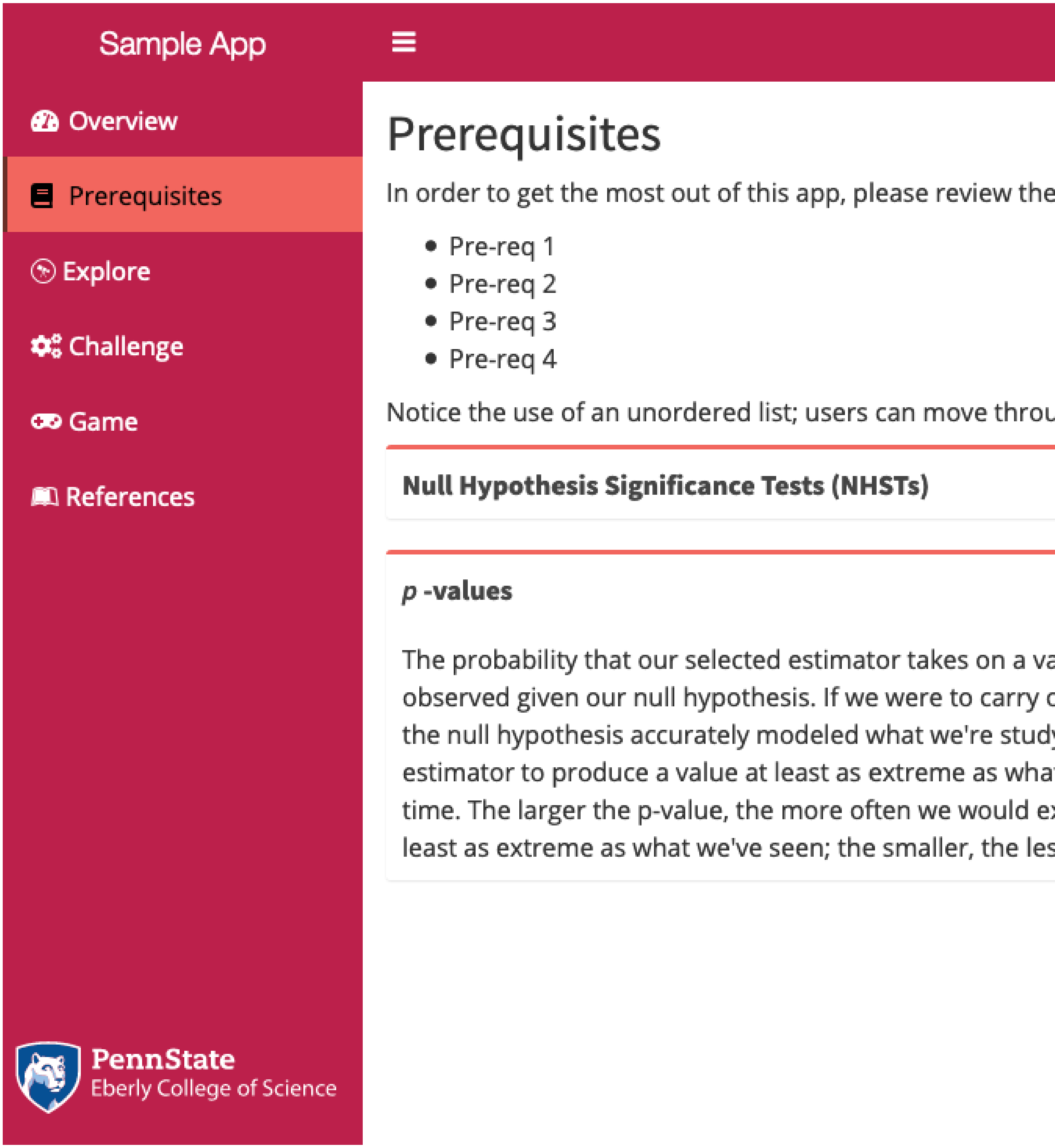
1. Review any prerequiste ideas using the Pre
2. Explore the Exploration Tab.
3. Challenge yourself.
4. Play the game to test how far you've come

Acknowledgements

This version of the app was developed and code
We would like to extend a special thanks to the S

Last Update:

Figure 8.22: Overview Page Using the Red Palette



Prerequisites

In order to get the most out of this app, please review the

- Pre-req 1
- Pre-req 2
- Pre-req 3
- Pre-req 4

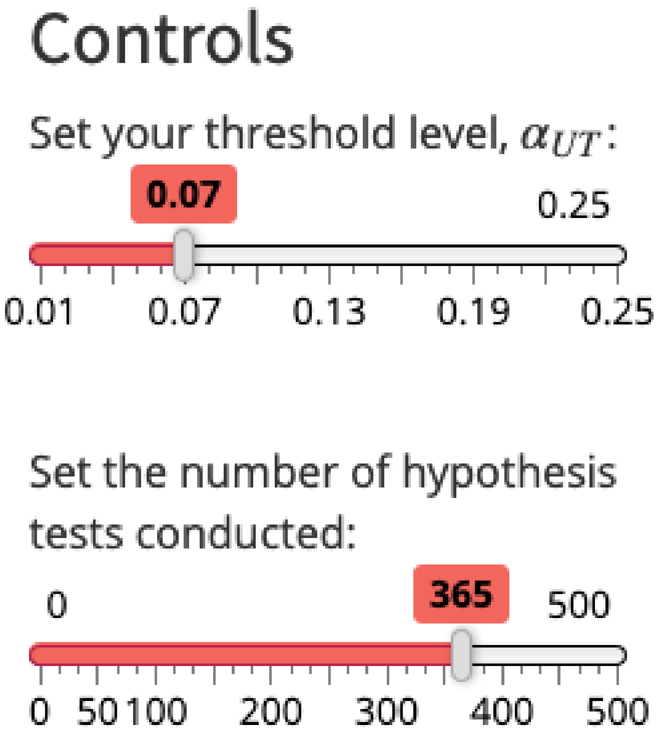
Notice the use of an unordered list; users can move throu

Null Hypothesis Significance Tests (NHSTs)

***p* -values**

The probability that our selected estimator takes on a va
observed given our null hypothesis. If we were to carry c
the null hypothesis accurately modeled what we're stud
estimator to produce a value at least as extreme as wha
time. The larger the p-value, the more often we would e
least as extreme as what we've seen; the smaller, the les

Figure 8.23: Collapsible Boxes Using the Red Palette



Controls

Set your threshold level, α_{UT}:

0.07

Set the number of hypothesis tests conducted:

365

Figure 8.24: Sliders Using the Red Palette

8.2 Dashboard Header

Each Dashboard Header contains only a couple of elements. The most important of these will be a [shortened] Title of your App. This will automatically be followed by the sidebar collapse/expand button. At the far right, you will then include a link to the home page of BOAST using the Home icon.

Additional icons might be included to the left of the Home icon. However, these icons remain the same for all Tabs/pages of your App and are thus are not appropriate for Tab/page specific information.

There should not be any additional elements in the Dashboard Header. Any links for navigate in your App should appear in the Sidebar on the left edge.

The width of the Title component of the Header should be 250; `titleWidth = 250`.

8.2.1 Creating the Dashboard Header

You will use the same structure regardless if you are using `app.R` or `ui.R`.

```
# [omitted code]
dashboardHeader(
      title = "Sample App", # You may use a shortened form of the title here
      titleWidth = "250",
      # the following is OPTIONAL
      tags$li(class = "dropdown", actionLink("info", icon("info"))),
      # You will see the following commented code in the Sample App;
      # you may delete this from your app
      # tags$li(class = "dropdown",
      #         tags$a(href='https://github.com/EducationShinyAppTeam/BOAST',
      #                icon("github"))),
      # the following is REQUIRED and must be last.
      tags$li(class = "dropdown",
              tags$a(href='https://shinyapps.science.psu.edu/', icon("home")))
)
# [omitted code]
```

A couple of things to notice:

- The `dashboardHeader` acts as a list environment, so it is safe to use `tags$li` here even though there isn't a `tags$ol` or `tags$ul`.
- There are not a lot of elements to the Header. We only add additional elements here if it is something static (i.e., un-changing) that is necessary for all pages of your App.
- You will need to set `class = "dropdown"` for each element after the `title` and `titleWidth`.
- The `info` icon is optional; if you use this keep in mind two things:
  - You will need to define what the link does in the `server` definition.

  - This link's action remains the same for all tabs of your app. Thus, you should not use this to store or be the main conveyance of critical information/instructions for using a particular tab of your App.
- The last element of the Header (i.e., the rightmost) will always be the home icon which takes the user to the BOAST home page.

8.2.2 What's Needed in the Server Definition

If you just have a bare bones Header (i.e., title and home button), you do not need to add anything special to your server definition.

If you do have something, for example, an Info button. You will need to add some additional code to your server definition. The code you add will depend upon the element. In general, for alert messages, we recommend that you use `shinyWidgets::sendSweetAlert`. Here's a generic example of what could be used for the Info button.

```r
# Required Packages
library(shinyWidgets)

# Move to your server definition
# Either look for server <- function(input, output, session) or open the server.R file

# [omitted code]

observeEvent(input$info, { # Replace "info" with the appropriate id
  shinyWidgets::sendSweetAlert(
    session = session, # This should stay as is
    title = "App Information",
    text = paste("[Message you want to give to the student]",
                 "Use paste with multiple lines to",
                 "improve code reability."),
    type = "info" # This option will depend upon the nature of your message
  )
})
```

The `type` argument will take one of several options. use the one that best aligns with your purposes.

- `info`: If you are just conveying some general information to your user, you'll use this value. This is the value that we will use in the vast majority of cases.
- `error`: Use this if your message tells the user that have committed some action that causes your App to fail.
- `question`: Use with an alert where you are asking the user to respond to a question.
- `warning`: If you want to give your user the opportunity to stop from doing something destructive (e.g., deleting data values), this would be

appropriate.
- `success`: Use this if your message let's the user know that something worked correctly.

Note: these buttons/links will throw an error when using the WAVE tool. This is expected at this time.

8.3 Dashboard Sidebar

The Sidebar is the main navigational tool for your users. Thus, if you want your user to access a certain tab/page, you must be sure to include this in the Sidebar. The only exception is if you are creating a series of levels inside the same Exploration, Challenge, or Game. These are referred to as Tabs Inside the Body and are covered in their own section.

The Sidebar should have a width of 250, (`width = 250`).

For full descriptions of each type of Tab, please refer to the Dashboard Body Section.

8.3.1 Creating the Dashboard Sidebar

This code will be the same regardless if you use `app.R` or `ui.R`.

```
# [omitted code]
dashboardSidebar(
  width = 250,
  sidebarMenu(
    id = "tabs",
    # Overivew is REQUIRED
    menuItem("Overview", tabName = "Overview", icon = icon("dashboard")),
    # Prerequisites is optional
    menuItem("Prerequisites", tabName = "Prerequisites", icon = icon("book")),
    # At least one of the next three is REQUIRED
    menuItem("Explore", tabName = "Explore", icon = icon("wpexplorer")),
    menuItem("Challenge", tabName = "Challenge", icon = icon("gears")),
    menuItem("Game", tabName = "Game", icon = icon("gamepad")),
    # References is REQUIRED
    menuItem("References", tabName = "References", icon = icon("leanpub"))
  ),
  # PSU Logo is REQUIRED and to be last
  tags$div(
    class = "sidebar-logo",
    boastUtils::psu_eberly_logo("reversed")
  )
)
# [omitted code]
```

Notice that after setting the `width` of the Sidebar, you will need to call the `sidebarMenu` function. This will create the appropriate structure for your Sidebar.

8.3.2 Sidebar Order

To ensure consistency across all apps, the Sidebar needs to have the following order:

1. The Overview Tab should always come first.
2. If used, a Prerequisites Tab will come second.
3. An Activity Tab (Explore, Challenge, or Game) will come next.
4. If there are multiple Activity Tabs, then
    a. Order by Concept so that Tabs dealing with the same idea are together,
    b. Then order by Explore, Challenge, then Game
    c. If a Game Tab covers multiple Explore/Challenge Tabs, place after the last of the Explore/Challenge Tabs
    d. For a set of Activity Tabs of the same type (e.g., three Explore Tabs), the order will be up to you and the learning goals for your App
5. The References Tab will always come after the Activity Tabs and be the last element of the `sidebarMenu`.
6. The PSU Logo will always be the last element of the Sidebar, and outside of the `sidebarMenu` call. Please refer to the Section 10.1.

Note: this order will also dictate how you should organize your code in the Dashboard Body.

8.3.3 Sidebar Names

There are three Tabs whose names are fixed (i.e., you can not change): the Overview, Prerequisites, and References. The Activity Tabs will be named in the following manner:

- If there is only one Activity Tab of each type, you may use the names Explore, Challenge, and Game.
- If there are multiple Tabs of each type, you will need to rename each tab as appropriate. For example,
    - In the NHST Caveats App, there are three Explore tabs: the Multiple Testing Caution, the Large Sample Caution, and the Small Sample Caution.
    - In the One-way ANOVA App there are two Games: a Matching Game and a Fill in the Blank game.

8.3.4 Sidebar Icons

Each type of Tab that appears in the Sidebar must use a specific icon:

- Overview – tachometer-alt (formerly called dashboard),
- Prerequisites – book,
- Explore Tabs – wpexplorer,
- Challenge Tabs – cogs (formerly called gears),

- Game Tabs – gamepad,
- References – leanpub,

If you come across any of these types of tabs that have a different icon or a missing icon, please create an issue in GitHub and/or fix.

If there is a type of tab that does fit these, please talk to Neil to see about what we need to add.

8.3.5 Submenus

Given the nature of our Apps, there is **NO** reason for having submenus. If you come across an app that has submenus or you believe that a submenu is necessary, then that is a good sign that you are looking at a "bloated" app. These apps need to be marked for review to investigate breaking the app in to two or more apps.

8.3.6 What's Needed in the Server Definition

If you have followed the above specifications, you do not need to add anything special to your server definition.

8.4 Dashboard Body

The Dashboard Body is where all content (text, images, plots, buttons, etc.) exists for the user to read, view, and interact with. Thus, this is the most important part of the layout of your App.

The order in which your code the Tabs in the Dashboard body needs to mirror the order of the tabs in the Sidebar. Thus, the first `tabItem` in the `dashboardBody` should be the Overview; the last, the References.

The Dashboard Body will begin with the following code in the UI section:

```r
# [code omitted]
dashboardBody(
  tabItems(
    tabItem(
      tabName = "Overview", # needs to match the names you used in the Sidebar
      withMathJax(), # if you need to display mathematics, include this line
      # [code for the tab]
    ),
    # repeat tabItem chunk for each subsequent tab
  )
)
# [code omitted]
```

The following subsections explain the purposes of each type of Tab.

8.4.1 The Overview Tab

This Tab is **REQUIRED** for all Apps. This is the main landing page of your App and should appear at the top of the Sidebar. The icon for this Tab must be "dashboard".

The Overview Tab must contain **ALL** of the following elements:

1. "Long/Formal App Title" (as Heading 1; this will be the **only** instance of Heading 1 in your App)
2. A description of the app (as paragraph text under the title)
3. "Instructions" (as Heading 2)
4. General instructions for using the App (using an Ordered List environment)
5. A button that will take the user to the next Tab/page (see Section 8.5.2 on buttons)
6. "Acknowledgements" (as Heading 2)
7. A listing of acknowledgements including, coders, content writers, etc. (as a paragraph)
8. Last Element: `div(class = "updated", "Last Update: mm/dd/yyyy by FL.")` with mm/dd/yyyy replaced with the date of the update you pushed to the server and FL replaced with your initials.

The purpose of the Overview Tab is the act like the front/home page of any newspaper, magazine, or website. Set the stage for what the user will be doing.

8.4.1.1 Creating the Overview Tab

Here's an example of making an Overview Tab

```
# Required Package
library(shinyBS)

# In the UI Section
# Inside the Dashboard Body

tabItem(
  tabName = "Overview",
  withMathJax(),
  h1("Sample Application for BOAST Apps"), # This should be the full name.
  p("This is a sample Shiny application for BOAST."),
  p("While not a proper app for helping students learn some statistical concept,
```

```r
    this app functions as an example for a variety of features."),
  h2("Instructions"),
  p("This information will change depending on what you want to do."),
  tags$ol(
    tags$li("Review any prerequiste ideas using the Prerequistes tab."),
    tags$li("Explore the Exploration Tab."),
    tags$li("Challenge yourself."),
    tags$li("Play the game to test how far you've come.")
    ),
  ##### Go Button--location and text will depend on your goals
  div(
    style = "text-align: center",
    bsButton(
      inputId = "explore1",
      label = "Explore!", # Notice there are NO spaces between the letters
      size = "large",
      icon = icon("bolt"),
      style = "default"
      )
    ),
  ##### Create two lines of space
  br(),
  br(),
  h2("Acknowledgements"),
  p("This version of the app was developed and coded by Neil J. Hatfield and Robert P.
    Carey, III.",
    br(),
    "We would like to extend a special thanks to the Shiny Program Students.",
    #### Create three lines of space
    br(),
    br(),
    br(),
    div(class = "updated", "Last Update: 5/13/2020 by NJH.")
  )
)
```

A few things to notice:

- If you need a new line but not a new paragraph, you use the `br()` tag.
- The label for the button needs to be in-line with Section 8.5.2.
- There should **NOT** be any spaces between letters of a button label. This is a violation of Accessibility as this destroys the label. While we might read "G (space) O" as the word "go", a screen reader reads out "gee" (pause) "oh" to the user.
- There is no need to use boldface or colons with the section headings when you properly use Heading tags. Thus, "Instructions:" does not follow this

Style Guide.

- There should not be an “About” heading. The text between the Title of your App and the Instructions head serves as the description.

8.4.1.2 What’s Needed in the Server Defintion

At bare minimum you will need to have one element in your server definition: the action for your button. If you have multiple buttons, you might need to have several more code chunks.

Here is a generic example for the button on the Overview Tab that moves the user to the appropriate next Tab:

```
## Define what each button does; repeat this style of coding for each button

# In your server section
# [code omitted]
observeEvent(
  eventExpr = input$expore1, #append the button's inputId to input$ as the event expre
  handlerExpr = { # This is the action portion of your button and must be in { }
  updateTabItems(session, # This how you allow the user to move Tabs
                 inputId = "tabs", # the id of your Sidebar
                 selected = "Explore" # Name of Tab to go to
                 )
})
```

In the rare cases where you’re having a button in the Overview Tab do something other than move to a new tab, you’ll change the `handlerExpr` to do that other action.

8.4.1.3 Buttons that Go to Activity Packets

An exception to the above is if the button is to allow the user to download/open up an activity file. In these cases, you’ll not place any code in the server definition. Rather, you’ll change the nature of the button. Specifically, you’ll not use `bsButton` but `actionButton`:

```
# In the UI Section
actionButton(
  inputId = "ap1",
  label = "Activity Packet",
  icon = icon("cloud-download"),
  onclick = "window.open('../../ActivityPackets/Caveats/')"
)
```

Notice that the `onclick` argument is what creates the action for button and references a relative path to a particular file. In this case a R Markdown file that lives the in directory called. (The actual file name isn’t listed to ensure that the processed/rendered version of the RMD is what the users see.)

8.4.2 A Prerequisites Tab

If your App needs to ensure that the user has the base understandings necessary to interact with your App, you'll need to create a prerequisites Tab. Otherwise, skip this Tab.

The icon for this Tab must be "book".

Use the word "Prerequisites" rather than "Pre-reqs", "Prereqs", or "Pre-requisites".

8.4.2.1 Types of Prerequisites

There are two different types of prerequisites: technical/conceptual and contextual. Both of these go into the Prerequisites tab.

Technical/Conceptual Prerequisites cover ideas that the user needs in order to fully engage with your App's statistical goal. For instance, if your App is about ANCOVA, the ideas of ANOVA and building a linear model would be good candidates for technical/conceptual prerequisites.

Contextual Prerequisites cover ideas that which are beneficial for the user to understand a context you're using. For example, if you are referencing an astragalus, you should include a brief explanation and/or picture of an astragalus.

Keep in mind that Contextual Prerequisites are different than context which should be part of the Activity Tab. If the information is necessary to interpret sliders/graphs and is *specific*, then you should include this information in the Activity Tab. If the information helps the user say "Oh, that's what they mean by [blank]", that is good sign of something to put in the Prerequisites Tab.

8.4.2.2 Text Links in Prerequisites (and Beyond)

In as many instances as possible, we would like to provide the user with a link to Online Notes of a World Campus Statistics course.

Note: what appears here is applicable any time you want to link to an webpage that is beyond BOAST.

The link that you provide must take the user to the appropriate location. Do not send the user to the home page for a course; rather, take them to the relevant page. To do this, you'll need to explore the Department of Statistics STAT ONLINE page and look through the courses.

You will create these links in-line, not as a button. Thus, they must be part of a paragraph block (i.e., inside a `p()` with other text) or as part of list item (i.e., inside a `li()`).

Your link must include descriptive text. Using "Click Here" is **not** descriptive. Rather say where the link will take the user. If you look through the links that we've included in this Style Guide, we've been modeling this. This descriptive text not only helps all users anticipate where they are going but also improves the accessibility of the links. (Plus, have you ever tried to click a small link on your phone?)

Once you find the appropriate page, you'll need to copy the URL for your link. There are some instances where we might be able to find an existing anchor (look for two inter-locking rings to appear when you place your cursor over a title) or make a request for adding an anchor. These are especially useful if what you want to link to is only part of the page.

*Note: not all requests for anchors may be fulfilled and not all course notes have anchors.

The styling of the link will be managed by the BOAST CSS file.

Here's are two examples of how you would code a text link:

```r
# [omitted code]
# Working in the UI section

# Example 1: in a paragraph
p("While not critical, you might wish to refresh your understanding on some of the basi
  of graphs in statistics. A good resource for this would be the ", # Notice the endin
  tags$a(
    herf = "https://online.stat.psu.edu/stat100/lesson/3/3.2#graphshapes", #the URL
    "STAT 100 Table of Graph Shapes" # the descriptive text for the link
    ),
  ". Feel free to check that resource out."
  # Notice the ending punctuation for the prior sentence is not part of the link.
)

# Example 2: in a list item
tags$ul(
  tags$li("Review the ", # Notice the ending space
    tags$a(
      herf = "https://online.stat.psu.edu/stat100/lesson/3/3.2#graphshapes", #the URL
      "STAT 100 Table of Graph Shapes" # the descriptive text for the link
      )
    # List items don't necessarily need ending punctuation.
    #Be consistent; either all items do or none.
  )
)
# [omitted code]
```

8.4.2.3 Creating a Prerequisites Tab

Here's an example of the code needed to create the Prerequisites Tab in the UI:

```
# [code omitted]
tabItem(
  tabName = "Prerequisites",
  withMathJax(), # this line only need if you display mathematics
  h2("Prerequisites"),
  p("In order to get the most out of this app, please review the following:"),
  tags$ul(
    tags$li("Pre-req 1"),
    tags$li("Pre-req 2"),
    tags$li("Pre-req 3"),
    tags$li("Pre-req 4")
    ),
    p("Notice the use of an unordered list; users can move through the list any way they wish."),
  p("A second style of doing prerequisites is with collapsible boxes:"),
  box(
    title = strong("Null Hypothesis Significance Tests (NHSTs)"),
    status = "primary",
    collapsible = TRUE,
    collapsed = TRUE,
    width = '100%',
    "In the Confirmatory Data Analysis tradition, null hypothesis significance tests serve as a
    critical tool to confirm that a particular theoretical model describes our data and to make a
    generalization from our sample to the broader population (i.e., make an inference). The null
    hypothesis often reflects the simpler of two models (e.g., 'no statistical difference',
    'there is an additive difference of 1', etc.) that we will use to build a sampling
    distribution for our chosen estimator. These methods let us test whether our sample data are
    consistent with this simple model (null hypothesis)."
  ),
  box(
    title = strong(tags$em("p"), "-values"),
    status = "primary",
    collapsible = TRUE,
    collapsed = FALSE,
    width = '100%',
    "The probability that our selected estimator takes on a value at least as extreme as what we
    observed given our null hypothesis. If we were to carry out our study infinitely many times
    and the null hypothesis accurately modeled what we're studying, then we would anticipate our
    estimator to produce a value at least as extreme as what we have seen 100*(p-value)% of the
    time. The larger the p-value, the more often we would expect our estimator to take on a value
    at least as extreme as what we've seen; the smaller, the less often."
  )
),
# [code omitted]
```

For more information on collapsible boxes, see Section 8.5.6.

8.4.2.4 What's Needed in the Server Definition

Generally speaking, the purpose of the Prerequisites Tab is to convey key background information for the user to double check they understand before moving into the heart of your App. Thus, this tab contains static text and images. You do no need to have anything in the server definition for the Prerequisites Tab.

8.4.3 Activity Tab(s)

The heart of your App is the one or more tabs where users interact with the App beyond simple navigation. These are the Activity Tabs. Some apps will have a single activity, others several; deciding on how many is part of the design process.

Currently, we have three types of Activity Tabs in BOAST:

- Exploration/Explore Tabs,
  - These tabs center around the user exploring the target concept.
  - These tabs will generally have more text on the page here than other types of Activity Tabs.
  - There are often guiding questions meant to help the user engage in productive explorations.
  - The goal is not to assess the user's understanding, but to support their construction of productive meanings for the concept.
- Challenge Tabs,
  - These tabs center around a user challenging themselves by testing out their understanding of a concept.
  - While there might still be a fair amount of text on the page, there will be less than an Exploration Tab.
  - Questions here will be in-between a guiding question and an assessment question.
  - The goal is to provide the user an opportunity to test and refine their understandings.
- Game Tabs,
  - These tabs center around the user review a concept (or several) in a game like format.
  - These generally have the least amount of non-question text on the page (i.e., instructions).
  - The goal is to provide an opportunity for a student to review and practice one or more concepts.

8.4.3.1 General Layout for Activity Tabs

While there are some differences between the different types of Activity Tabs, there is one firm constant for all of them:

Each Tab should contain all information/instructions for the user to be able to interact with the activity without having to switch to other

Tabs.

There is nothing worse for the user than getting to an Activity Tab and not knowing what they are supposed to do. The Instructions on the Overview are for using the *entire* App, not any one particular page. Thus, you need to have specific page instructions somewhere on the page.

Keep in mind that this is beneficial to all users, but especially those who are using assistive technology such as screen readers. If you put your instructions on a separate tab, you are now requiring that your user memorize those instructions. This is already cognitively demanding for sighted individuals, but for users with vision impairments, even more so.

In addition, we will adopt a 3-part layout:

- Across the top will be any general information the user needs to interact with your App as well as an context information.
- To the left and wrapped in a well panel will be the inputs/controls that the user will need to manipulate.
- To the right and NOT in a well panel will be the outputs (graphs, images, R output)

There will be some cases where this general layout does not necessarily work. For instance, Tic-Tac-Toe games will not follow this layout.

8.4.3.2 UI and Server Definitions

At this time, we do not have any examples of creating these tabs for this Style Guide. Rather, we encourage you to look at the many examples in the Book of Apps for Statistics Teaching as well as the GitHub Repository for BOAST.

Keep in mind that of all tabs in your App, these tabs will demand the most for the UI and the Server.

8.4.4 References

The last Tab will be for your references. This Tab is **REQUIRED** and is where you will place a reference list for all of the following items that you used in your app:

- All `R` packages you used
- Sources of any Code you used directly or drew heavily upon from other people
- Pictures and/or other images
- Data sets
- Refer to the Chapter 14 on Documentation of this Style Guide for more information.

The icon for this Tab must be “leanpub”, .

We will additionally place licensing information for your App at the bottom of the Reference page. We have created a function in `boastUtils` (version 0.1.6.1) that will automatically put the correct information on the page.

8.4.4.1 Creating a References Tab

Creating a References Tab mimics both the Overview and Prerequisites Tab structure and is done in the UI section:

```
# In the UI Section
# [code omitted]
tabItem(
  tabName = "References",
  withMathJax(), # Rarely, if ever, will you need MathJax in the references
  h2("References"),
  p( # Each reference is in its own paragraph
    class = "hangingindent", # you must set this class argument
    "Bailey, E. (2015), shinyBS: Twitter bootstrap components for shiny, R package. Ava
    from https://CRAN.R-project.org/package=shinyBS"
    ),
  p(
    class = "hangingindent",
    "Carey, R. (2019), boastUtils: BOAST Utilities, R Package. Available from
    https://github.com/EducationShinyAppTeam/boastUtils"
    ),
  p(
    class = "hangingindent",
    "Chang, W. and Borges Ribeio, B. (2018), shinydashboard: Create dashboards with 'Sh
    Package. Available from https://CRAN.R-project.org/package=shinydashboard"
    ),
  p(
    class = "hangingindent",
    "Chang, W., Cheng, J., Allaire, J., Xie, Y., and McPherson, J. (2019),  shiny: Web
    application framework for R, R Package. Available from
    https://CRAN.R-project.org/package=shiny"
  ),
  p(
    class = "hangingindent",
    "Hatfield, N. J. (2019), Caveats of NHST, Shiny Web App. Available from
    https://github.com/EducationShinyAppTeam/Significance_Testing_Caveats/tree/Pedagogi
  ),
  p(
    class = "hangingindent",
    "Wickham, H. (2016), ggplot2: Elegant graphics for data analysis, R Package, New Yo
    Springer-Verlag. Available from https://ggplot2.tidyverse.org"
  ),
  br(), # Three blank spaces
  br(),
  br(),
  boastUtils::copyrightInfo()
)
# [code omitted]
```

8.4.4.2 What's Needed in the Server Definition

You do not need to place anything in the server definition for the References Tab.

8.4.5 Tabs Inside the Body

There are two types of tabs in a Shiny app: there are the `tabItem` (i.e., the pages within an app and should appear in the Sidebar) and `tabPanel` (i.e., creating sub-pages or independent sections). In this section, we will discuss this later case.

Deciding on whether to use `tabPanel` is going to depend on several things:

1. Do you have two or more aspects that are related enough that they shouldn't be their own separate tabs/pages of your App?
   a. If NO, then you shouldn't use `tabPanel`.

   b. If YES, then continue.
2. Are any of your aspects something that would be better suited as a Challenge or Game tab?
   a. If YES, move that aspect to a separate page. If you still have 2+ aspects, continue.

   b. If NO, continue.

3. Are the aspects independent enough that a person can skip a couple and still use the App successfully?
   a. If NO, then you should re-consider your design.

   b. If YES, then proceed with using `tabPanel` in you design.

When you go to make a set of tab panels you will need to first create a `tabsetPanel` which will wrap around all of the individual panels. Use `type = "tabs"`.

The tabs inside the body should automatically appear horizontally and along the top of the tab body (i.e., in the white space below the Dashboard Header). Any visual styling will be managed by the BOAST CSS file at a global level.

8.5 Common Elements

In addition to the Dashboard elements of the apps, there are other elements that are common. This include things such as how inputs should be ordered, buttons, correct/incorrect indicators, and animation buttons.

For information about popovers, rollovers, hover text, or tool tips, please see Section 13.2.

8.5.1 Ordering Inputs

One of the most powerful aspects of Shiny apps is that the user interacts with them. Thus, we need to consider not only the ways in which user interact (e.g., buttons, sliders, text entry, etc.) but also the order in which you want the user to manipulate the inputs. Coming up with a single declaration for how to order inputs in all cases is not necessarily feasible. However, we can set up a general guideline for how to make decisions on ordering your inputs.

Please use the following guidelines for determining the order of inputs in the User Interface (UI):

1. In general, if you want your user to do things in certain order, make your inputs appear in that order. For example, If you want them to pick a data set, then an unusualness threshold/significance level, what attribute to test, and then set a parameter value, then your inputs should appear in that order.
2. Make use of how we read the English language, i.e., Top-to-Bottom and Left-to-Right to provide an implicit ordering for your user.
3. If a user needs to carry out steps in particular sequence for your App to run properly, then place your inputs inside of an Ordered List environment with explicit text on what they should do. For example,
    1. Choose your data set: [dropdown]

    2. Set your unusualness threshold/significance level
       [slider]

    3. Which attribute do you want to test: [dropdown]

    4. What parameter value do you want to use: [numeric input]
4. If an input is going to reset other inputs you should either:
    a. Warn the user before hand

    b. Move the input to the top of the list

    c. Program the input to not reset other inputs, or

    d. Some combination of the above

5. If the inputs are not dynamically linked to the output (e.g., plots automatically update with a change in the input's value), then you should include a button that says "Make Plot" at the end of the inputs.

8.5.2 Buttons

Buttons are one way in which users interact with the apps. The two most common button functions that we use are `shiny::actionButton` and `shinyBS::bsButton`. Both functions share many of the same features. Two ways in which they are different is that `shinyBS::bsButton` has an additional `style` argument while `shiny::actionButton` has a `width` argument that gives you fine grain size control (`bsButton` just has a qualitative size option).

There are three key styling aspects to every button: shape/animation, color, and text & icon.

8.5.2.1 Shape/Animation

All shape aspects of buttons will be controlled by CSS. The standard shape will be rectangular (the default). Sizing will be controlled by CSS although setting `size = "large"` for the `bsButton` call may be done.

We have a number of apps where a button will change shape/size when a person hovers their cursor over it. This "animation" is to be discontinued. This is to say that buttons which change shape/size should be flagged as issues and resolved at the first opportunity.

At most, the button's color might change (i.e., lighten or darken only), depending on the context.

8.5.2.2 Color

The coloring of the button will also be controlled by CSS in one of two ways.

The default way will be through the BOAST CSS. This will ensure that the selected color scheme for your App will be consistent.

The second way only applies to `bsButton` and the `style` argument. Here, this option references an external CSS file beyond BOAST. We see these most often in Game Activity Tabs. Use the following list to guide you in choosing which style is appropriate:

- `warning`: Good for when you want the user to proceed with caution; for example a submit button in a game.
- `danger`: Good for when you want the user to think twice before clicking; for example, a reset game button.
- `success`: Good for when you want to convey that the user can proceed safely; for example, a button that advances the user through the game

- `info`: Good for when you want to give some additional information; for example, a button that triggers game instructions popping up, a button that gives a hint, or a button that might filter a question pool.

When in doubt, use the the `default` style option (or even omit this argument) for `bsButton` or use `actionButton`.

8.5.2.3 Text & Icon

The last styling element of a button is two-fold: the text that is in the button and the icon.

Here are some guidelines for text of a button:

- All buttons must have some text.
- Generally speaking, the text should be relatively brief and clear.
  - Don't use "Go to the next page" when you could use "Next"
- The text should make sense with the action of the button; for example,
  - "Reset" if the button resets something (a game, a plot, inputs)
  - "Submit" if the button triggers the app to grab and process input values
  - "Make Graph" if button causes a graph to be generated
  - "Show/Hide Graph" if a button makes a graph object appear/disappear
  - "Next" if a button moves the user along some path.
- If the button references something like a particular tab (prerequisites, exploration, etc.), the text should reflect this.
  - "Explore!" for a button that takes a user to an Exploration tab.
  - "Prerequisites" for a button that takes a user to a Prerequisites tab.
  - "Challenge Yourself!" for a button takes a user to a Challenge tab.
  - "Play!" for a button that takes a user to Game tab.
- If a button references an object like an activity packet or a download prompt the text should refer to that
  - "Activity Packet" for a button that would open up and/or download a packet for the user
  - "Download Data" for a button that would download a data file.
- Clarity is essential. If there are multiple buttons on the page, make sure that you use clear text for what button does and/or references.

Here are guidelines for the inclusion of icons in a button:

- Game buttons will NOT have any icons.
- Direction Buttons (e.g., "Next" or "Previous") will NOT have any icons. Rather make the button text "« Previous" or "Next »"
- A "Prerequisites" button will use the "book" icon,
- All other tab buttons (labels ending with "!") will use the "bolt" icon,
- A download button will use the "cloud-download-alt" icon,

8.5.3 Correct/Incorrect Marks

In games, you can give the user a visual cue as to whether they are correct or incorrect through the use of two images:

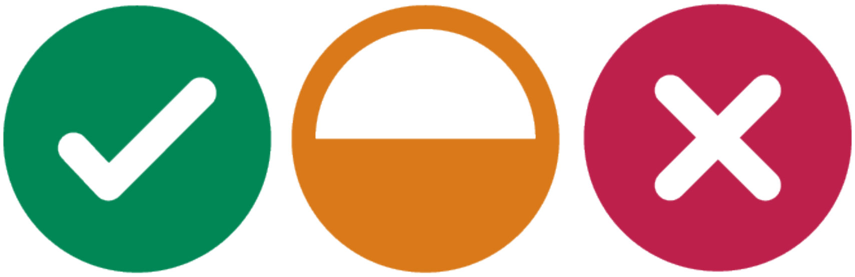


Figure 8.25: Correct, Partially Correct, and Incorrect Marks

You can save these two images by right-clicking on them and selecting "Save Image As…". You will need to put them in the `www` folder/directory of your App.

Their placement in your App will depend upon what makes the most sense.

Be sure that you add alternative text to these images; see Section 10.4.6. Do not use "right" for alt text; use "correct".

8.5.4 Animation Buttons

One feature of slider inputs is the option to include a Play/Pause button that allows the user to create an animation of your plot. Enabling this option can be quite useful if allowing the user to move through the whole set of slider values is desirable.

To enable this, you'll need to make use of the `animate` argument:

```
#[code omitted]
sliderInput(
  inputId = "mtcAlpha",
  label = "Set your threshold level, \\(\\alpha_{UT}\\):",
  min = 0.01,
  max = 0.25,
  value = 0.1,
  step = 0.01,
  animate = animationOptions(
    interval = 1000, loop = TRUE))
#[code omitted]
```

You can set `animate=TRUE`, `animate=FALSE` or invoke the `animationOptions` function as we've done in the example and recommend. This will force you to make some important decisions: namely, how long the slider should wait between each movement (`interval`, in milliseconds) and should the animation start over once the slider reaches the maximum (`loop`).

The `interval` is going to the most challenging value to figure out. This timer ***ignores*** everything else; that is, it doesn't wait to see whether your plot has updated. Remember, the more complicated the process that generates your plot is, the longer your App will need to render the plot. Thus, you can quickly get into a case where the slider has advanced several times while your App is still trying to render the first update. While `renderCachePlot` can help speed things up, keep in mind that you still might need to play around with the `interval` value to ensure smooth functionality.

Make sure when you're testing an animated slider to vary all of the parameters involved in the graph. This will help ensure that you test adequately.

The styling of the play/pause button will be controlled by the BOAST CSS file.

8.5.5 Progress Bar

Consider adding a loading bar to show the process for intense computations; this will help the user understand that your App is processing and not frozen/broken.

8.5.6 Collapsible Boxes

One technique that we can make use of to cut down on the amount of visible text on a page is through the use of collapsible boxes. Collapsible boxes are preferred than other methods in that 1) the content remains on the page and thereby accessible, and 2) the user retains control for when to show/hide this information.

Two great places to consider using collapsible boxes include the Prerequisites Tab and **static** Context Information across the top of an Activity Tab.

*Note: if the context information changes (e.g., can be switched due to user actions), then collapsible boxes should **NOT** be used. The context in these cases should **always** remain visible.*

8.5.6.1 Creating Collapsible Boxes

To create a Collapsible Box, you'll need to work in th UI section of your code:

```r
# [code omitted]
box(
  title = strong("Title for the Box"), # Use the strong tag
  # Give either the title of the review concept or "Context"
  status = "primary", # Leave as primary
  collapsible = TRUE, # This allows collapsing
  collapsed = FALSE, # Initial value
  # If the only collapsible item, use FALSE
  # If there are multiple, the first one is FALSE, others can be set to TRUE.
  width = '100%', # use this setting
  "The text that will 'disappear' goes here."
),
# [code omitted]
```

Given the static nature of the information in a collapsible box, you should not need to add anything to the server definition.

The styling of Collapsible Boxes is controlled by the central CSS file. If you use the above code as your template, the coloring will automatically match your App's assigned color theme.

8.5.7 Well Panels

Well panels are visual styling that we use to help offset user controls (i.e., inputs) from both context and graphs (i.e., outputs).

To place something inside of a well panel, all you need to do is wrap that object in the `wellPanel` function. However, the placement of the `wellPanel` call matters.

If you wrap each individual element in `wellPanel` you'll get a separate well panel for each call. If you place `wellPanel` too high up in your code, you'll end up putting everything into the well panel.

```
# In the UI Section
# [code omitted]
# Inside a tabItem
fluidRow( # this allows you to create dynamic columns for responsive design (mobile friendly)
  column( # this is one of those columns
    width = 4, # the initial grid width; all columns must add to 12
    wellPanel( # Placing the wellPanel call here will wrap around all controls
      h3("Controls"),
      sliderInput(
        inputId = "mtcAlpha",
        label = "Set your threshold level, \\(\\alpha_{UT}\\):",
        min = 0.01,
        max = 0.25,
        value = 0.1,
        step = 0.01,
        animate = animationOptions(interval = 1000, loop = TRUE)
        ),
      br(), # creating vertical space between sliders
      sliderInput(
        inputId = "mtcTests",
        label = "Set the number of hypothesis tests conducted:",
        min = 0,
        max = 500,
        value = 5,
        step = 5
        )
      ) # this closes the wellPanel
    ), # this closes the first column
  column(
    width = 8,
    h3("Plot"),
    plotOutput("pplotMTC"),
    bsPopover(
      id = "pplotMTC",
      title = "Investigate!",
      content = "What happens to the number of statistically significant tests when you
      increase the number of tests?",
      placement = "top"
      )
    ) # closes second column
  ), #closes the fluid row
```

Chapter 9

App Layout

When considering the Visual Appearance of your App, there are two major areas of consideration: the Layout and what we refer to as Design Styling. Each of these will be handled in turn. Keep in mind that these two aspects are interrelated and have significant cross-over.

In this chapter, we will be focusing on the layout aspect of how your App looks. That is to say, we're going to talk about the standards/guidelines that cut across all of our apps so that they look like they belong together.

A good number of the elements of the layout will have a direct consequence on your coding. For instance, in the section on Organizing Your Code, you saw that the UI definition needed to come before the Server definition. This section defines additional organizational structure to your code as it pertains to the layout of your App.

As a reminder, one of the most important benefits of using the `boastApp` function from the `boastUtils` package is that is that certain aspects of Visual Appearance will be automatically handled for you. However, you still need to do adhere to this Style Guide.

9.1 Dashboard

All apps will make use of a Dashboard structure. This divides the visual appearance of each App into three main areas.

- Across the top of the App will be the Header
- Along the left side of the App will be the navigation list (the Sidebar) where the various Tabs (pages) of your App will be listed
- The last area is the Body; this is where all content will appear

Several of the older apps will have outdated UI calls including, but not limited to: `shinyUI` and `navbarPage`. These functions should no longer be used; apps that use them need to be updated to become compliant with this Style Guide.

9.1.1 Creating the Dashboard

Creating the overarching Dashboard layout in your App is actually quit easy:

```r
# Required package
library(shinydashboard)

# For app.R files
# [code omitted]
ui <- list(
  dashboardPage(
    skin = "blue",
    # [code omitted]
  )
)

# For ui.R files
dashboardPage(
  skin = "blue",
  # [code omitted]
)
```

The `dashboardPage` function will wrap around the rest of your UI element, thereby establishing the overarching structure. Three of this function's arguments (`header`, `sidebar`, and `body`) will be covered in following sections. We do not need to worry about the `title` argument as this will be controlled by the Dashboard Header.

The one argument that you need to explicitly set for the `dashboardPage` is the `skin` argument. This argument sets the overall color theme for your App. While this is something that is more an aspect of Design Style, your only opportunity to set this value is here with the Layout.

9.1.2 Color in the User Interface

Within BOAST, we use color themes to help provide consistency for the elements of each app and to denote different chapters. Part of the standardization process of this Style Guide seeks to bring the many fractured color themes together into a cohesive, centrally managed set. This helps reduce the programming burden on the students, who should focus on the R side of the programming, not the CSS side.

All aspects of color in the User Interface should be controlled through the CSS file(s). This includes all of the following:

- Dashboard coloring (Header, Sidepanel, Body)
- Text coloring
- Coloring of Controls (including buttons, sliders, and other input fields)

By using CSS, especially through `boastApp`, you'll be able to ensure that there is consistent coloring throughout your App.

9.1.2.1 Implementing a Color Theme

To activate a color theme is a simple process, especially if you are following this Style Guide and using the `boastUtils` package. (If you are in an App using ui.R and server.R, make sure that the boast.css call is in the ui.R file. See Section 5.2.4.)

In your App's code, go to where you first call the function `dashboardPage`. Then, as the first argument you'll type `skin = "[theme]"` before moving on the next argument, `dashboardHeader`.

You will replace `[theme]` with one of the following: `blue`, `green`, `purple`, `yellow`, `red` or `black`. The choice will be determined by the color assigned to that chapter. This is all you have to do.

If you are unsure what color to put, use `blue` as the default.

9.1.2.2 The Themes

There are six color themes that we've currently made. The names of the themes are a general indication of coloring, with one exception. The `black` theme is not black but rather an aqua/teal set. The themes are typically three colors (four for `blue`) and based upon the Penn State Palettes. Non-Penn State colors will be denoted with asterisks.

All of the themes have been checked against 8 different forms of color blindness.

9.1.2.2.1 Blue The Blue Palette is our central palette and should be used by default. The Blue Palette looks like the following:

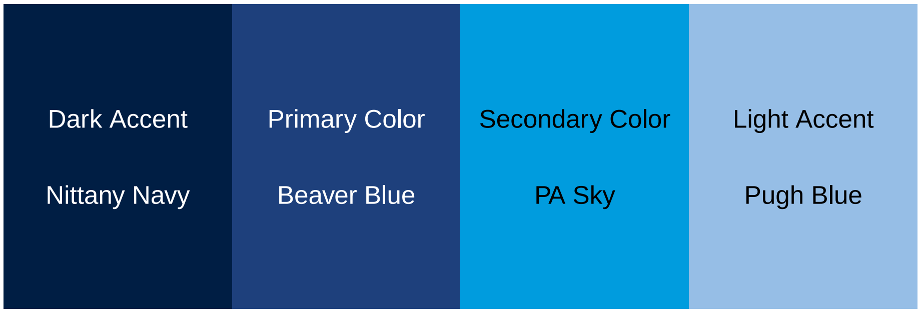


Figure 9.1: The Blue Palette

Here is what the Blue Palette looks like in action:

9.1.2.2.2 Green The Green Palette looks like the following:

Here is what the Green Palette looks like in action:

9.1.2.2.3 Purple The Purple Palette looks like the following:

Here is what the Purple Palette looks like in action:

9.1.2.2.4 Black The "Black" Palette is not pegged to the color black, but rather teal/aqua colors. However, to call the theme in the Shiny dashboard, the user must use the value `black` for the the `skin` argument. Here's what the "Black" Palette looks like:

Here is what the "Black" Palette looks like in action:

9.1.2.2.5 Yellow The Yellow Palette is still under consideration. The current set looks like the following:

Here is what the Yellow Palette looks like in action:

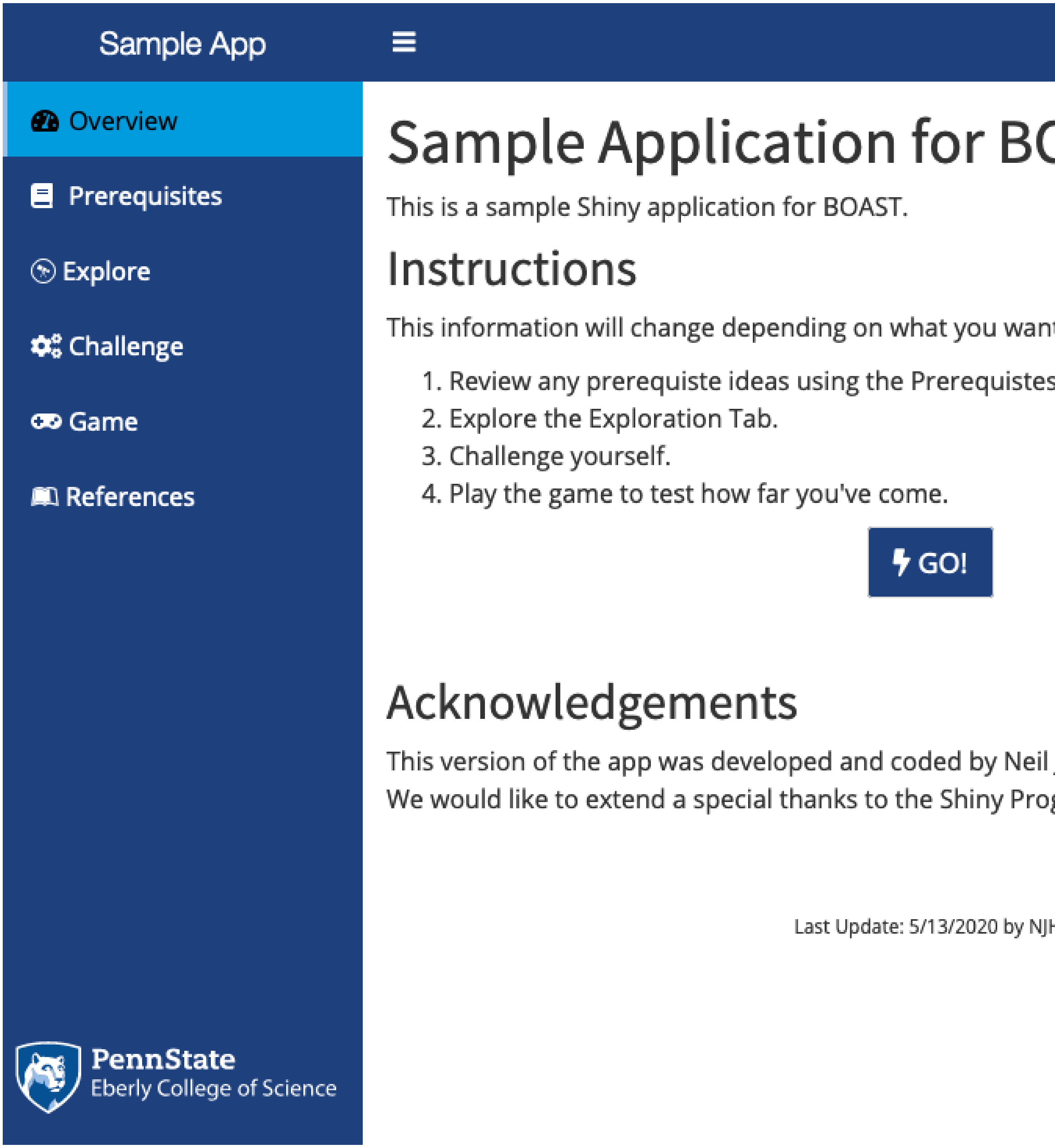
9.1.2.2.6 Red The Red Palette is still under construction. Here's the current set:

Here is what the Red Palette looks like in action:

9.1.3 Current Chapter Color Assignments

Here are the current (05/27/2020) color theme assignments for chapters:

- Chapter 1: Data Gathering RED
- Chapter 2: Data Description YELLOW
- Chapter 3: Basic Probability BLUE
- Chapter 4: Statistical Inference PURPLE
- Chapter 5: Probability BLUE
- Chapter 6: Regression "BLACK"
- Chapter 7: ANOVA "BLACK"
- Chapter 8: Time Series PURPLE
- Chapter 9: Sampling RED
- Chapter 10: Categorical Data YELLOW
- Chapter 11: Data Science GREEN
- Chapter 12: Stochastic Processes BLUE
- Chapter 13: Biology GREEN



Sample Application for BC

This is a sample Shiny application for BOAST.

Instructions

This information will change depending on what you want

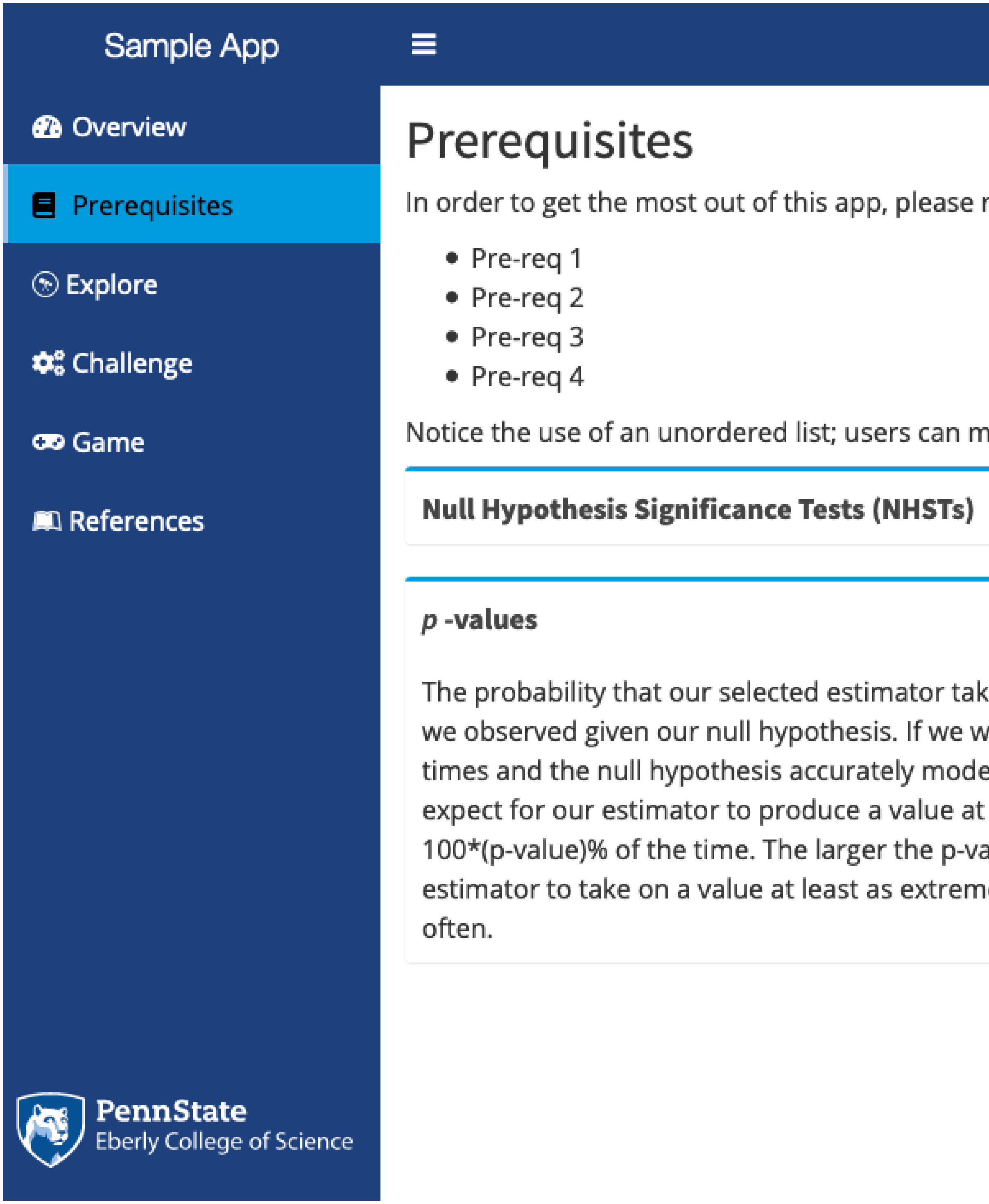
1. Review any prerequiste ideas using the Prerequistes
2. Explore the Exploration Tab.
3. Challenge yourself.
4. Play the game to test how far you've come.

Acknowledgements

This version of the app was developed and coded by Neil
We would like to extend a special thanks to the Shiny Prog

Last Update: 5/13/2020 by NJH

Figure 9.2: Overview Page Using the Blue Palette



Prerequisites

In order to get the most out of this app, please r

- Pre-req 1
- Pre-req 2
- Pre-req 3
- Pre-req 4

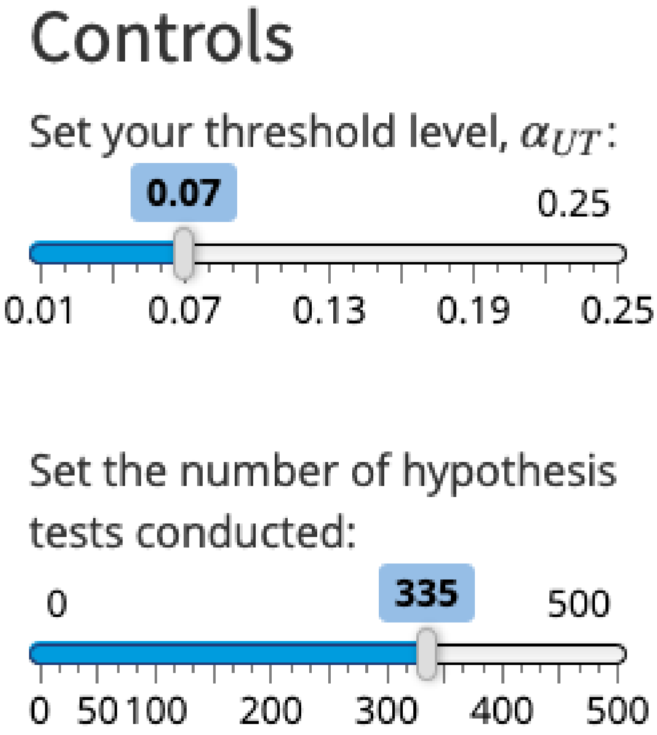
Notice the use of an unordered list; users can m

Null Hypothesis Significance Tests (NHSTs)

***p* -values**

The probability that our selected estimator tak
we observed given our null hypothesis. If we w
times and the null hypothesis accurately mode
expect for our estimator to produce a value at
100*(p-value)% of the time. The larger the p-va
estimator to take on a value at least as extrem
often.

Figure 9.3: Collapsible Boxes Using the Blue Palette



Controls

Set your threshold level, α_{UT}: 0.07

Set the number of hypothesis tests conducted: 335

Figure 9.4: Sliders Using the Blue Palette

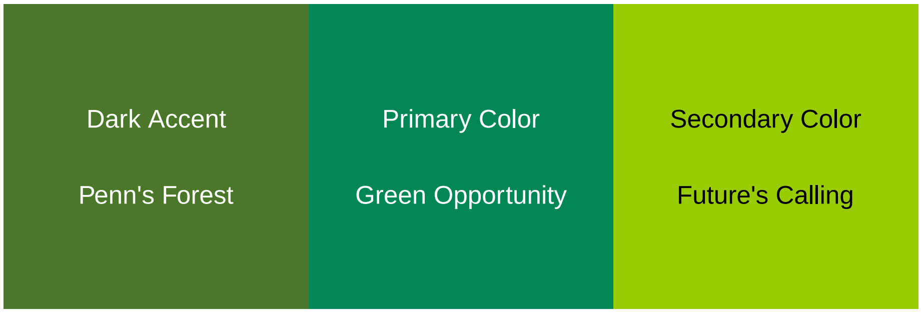
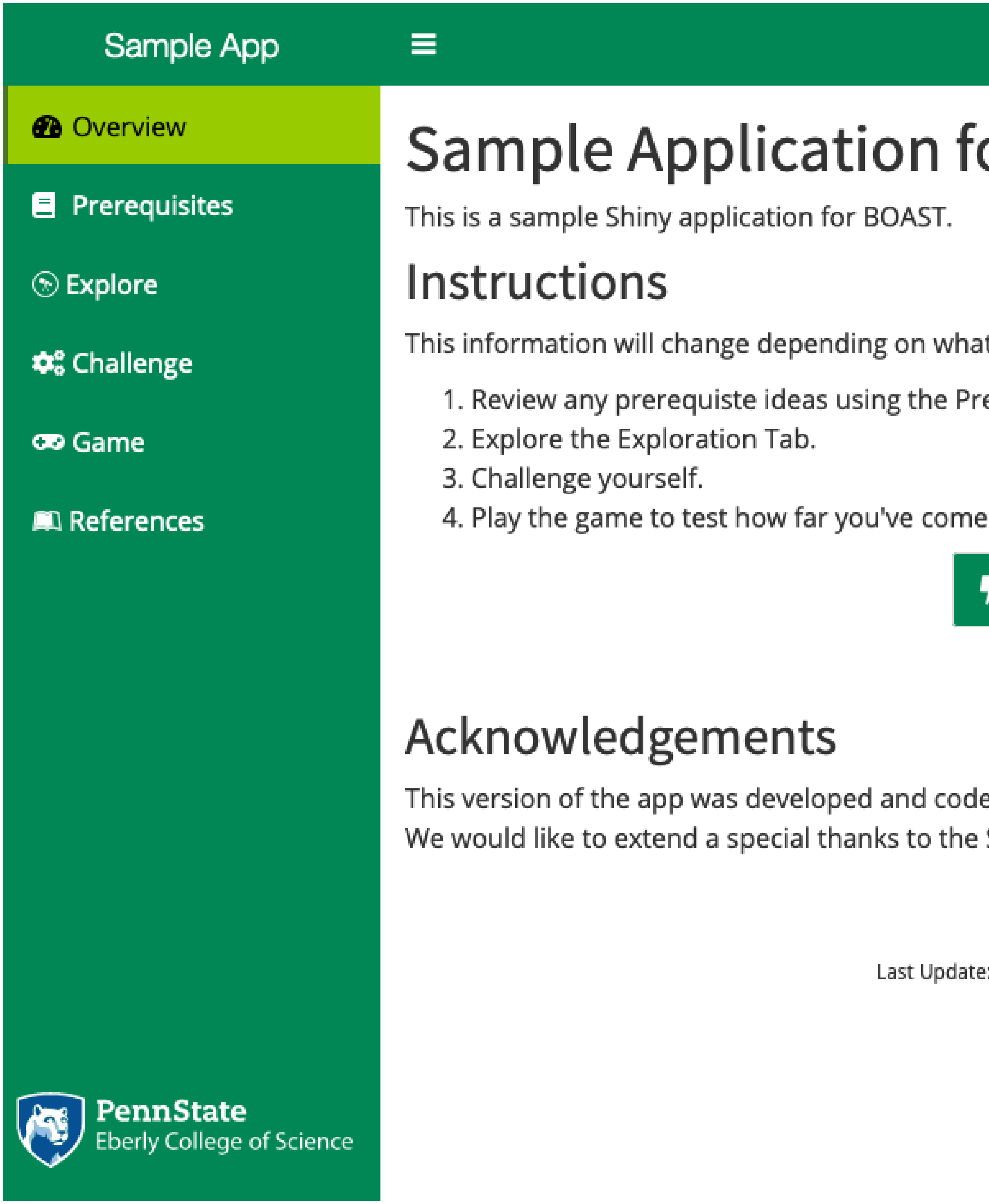


Figure 9.5: The Green Palette



Sample Application fo

This is a sample Shiny application for BOAST.

Instructions

This information will change depending on what

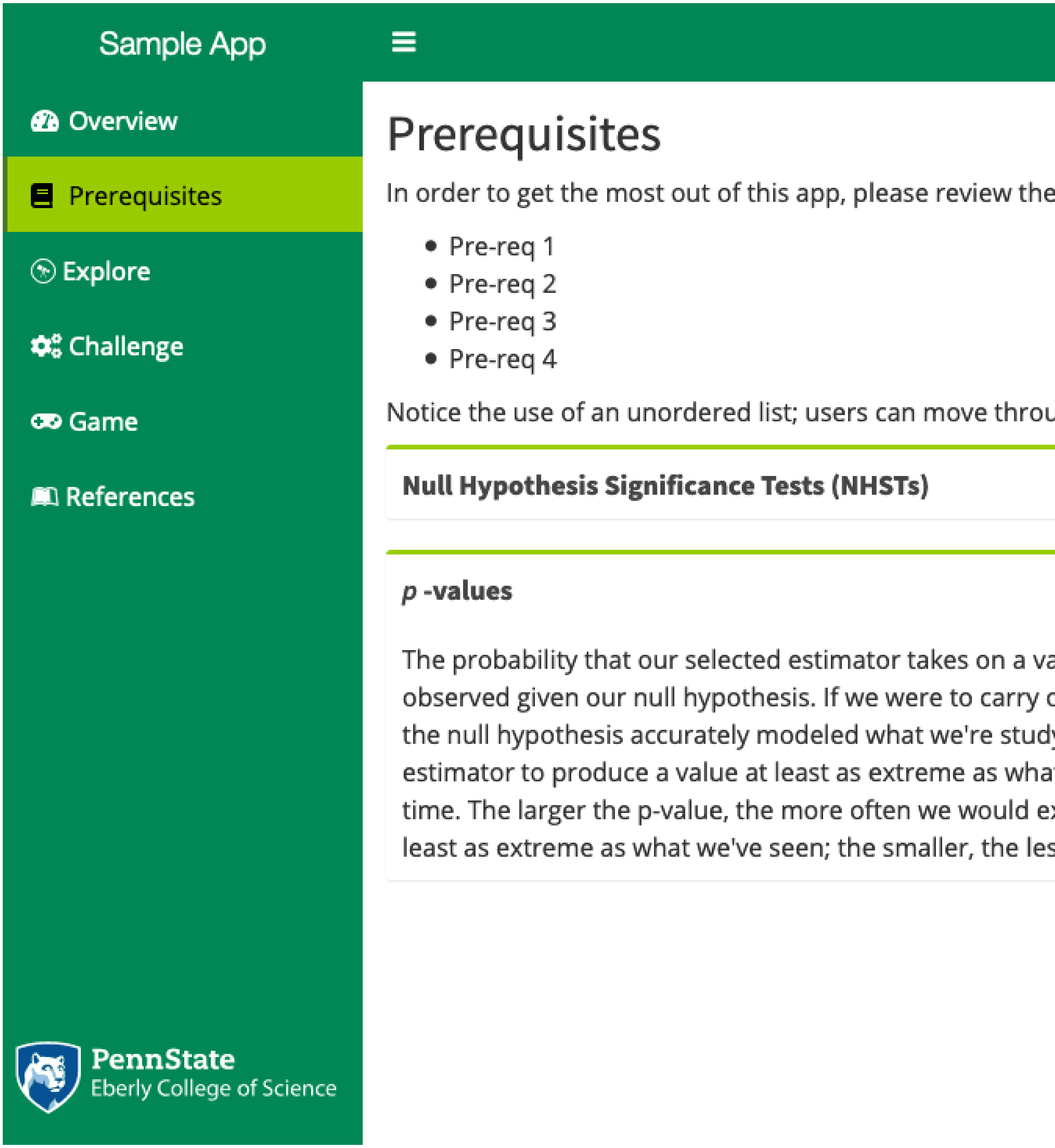
1. Review any prerequiste ideas using the Pre
2. Explore the Exploration Tab.
3. Challenge yourself.
4. Play the game to test how far you've come

Acknowledgements

This version of the app was developed and code
We would like to extend a special thanks to the

Last Update

Figure 9.6: Overview Page Using the Green Palette



Prerequisites

In order to get the most out of this app, please review the

- Pre-req 1
- Pre-req 2
- Pre-req 3
- Pre-req 4

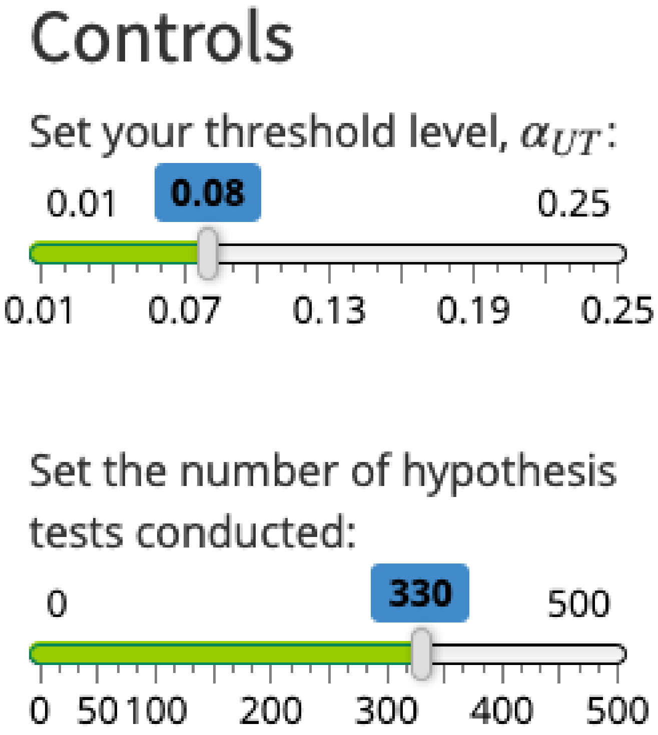
Notice the use of an unordered list; users can move throu

Null Hypothesis Significance Tests (NHSTs)

***p* -values**

The probability that our selected estimator takes on a va observed given our null hypothesis. If we were to carry c the null hypothesis accurately modeled what we're study estimator to produce a value at least as extreme as what time. The larger the p-value, the more often we would ex least as extreme as what we've seen; the smaller, the les

Figure 9.7: Collapsible Boxes Using the Green Palette



Controls

Set your threshold level, α_{UT}:

0.08

Set the number of hypothesis tests conducted:

330

Figure 9.8: Sliders Using the Green Palette

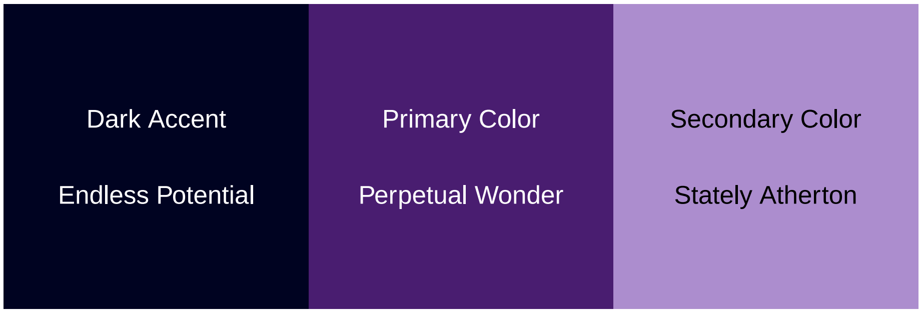
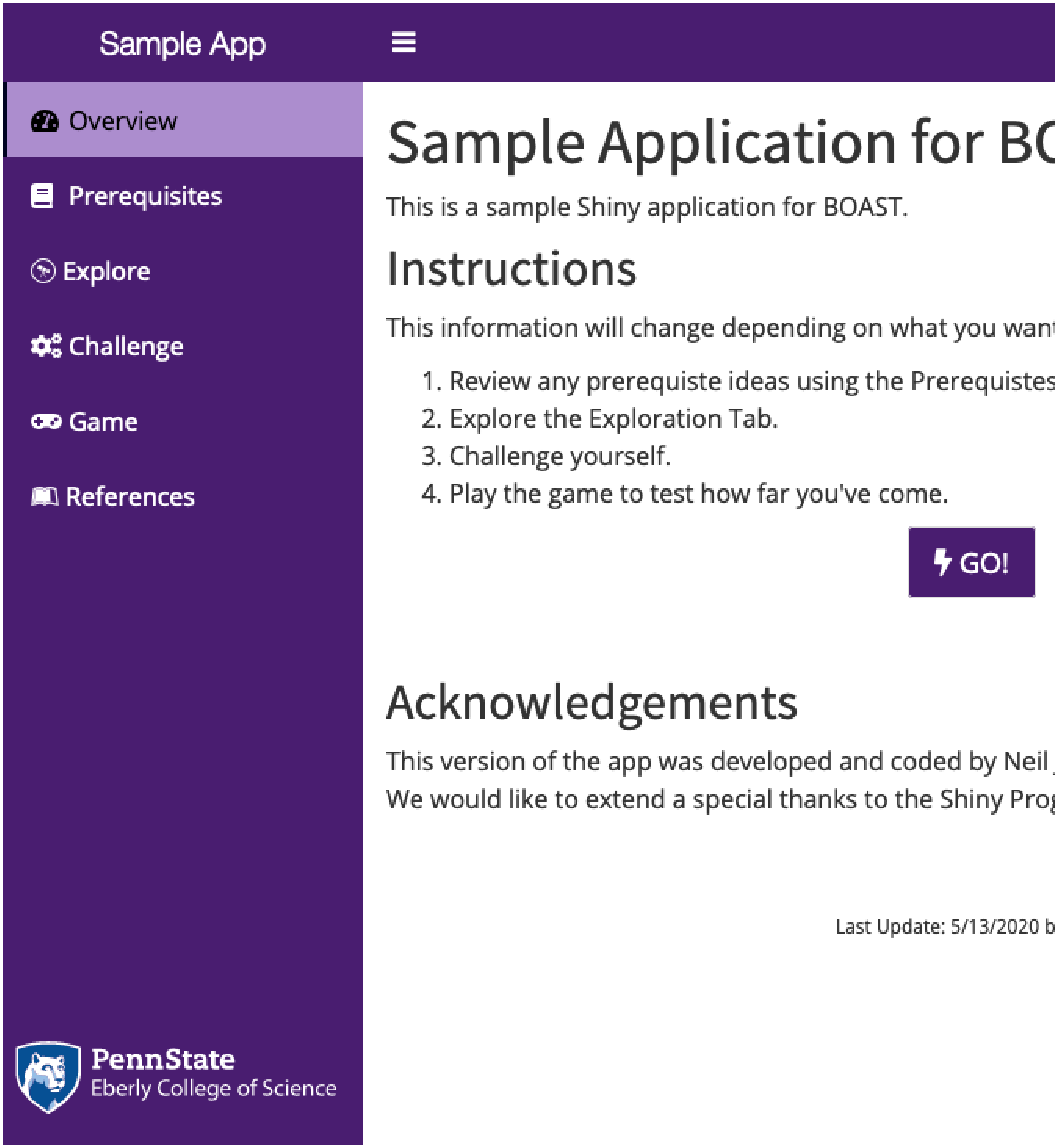


Figure 9.9: The Purple Palette



Sample Application for B

This is a sample Shiny application for BOAST.

Instructions

This information will change depending on what you wan

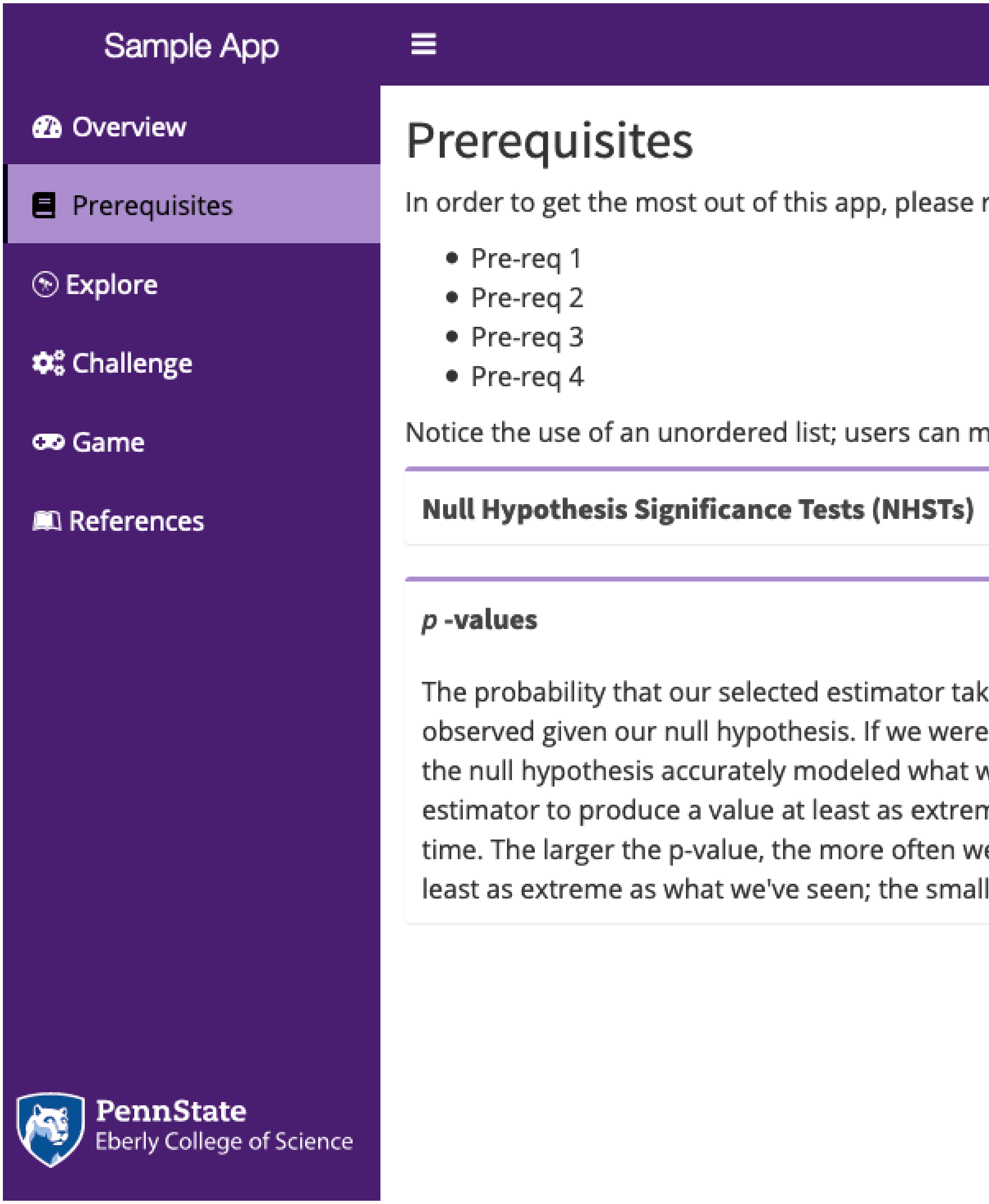
1. Review any prerequiste ideas using the Prerequistes
2. Explore the Exploration Tab.
3. Challenge yourself.
4. Play the game to test how far you've come.

Acknowledgements

This version of the app was developed and coded by Neil
We would like to extend a special thanks to the Shiny Prog

Last Update: 5/13/2020 b

Figure 9.10: Overview Page Using the Purple Palette



Prerequisites

In order to get the most out of this app, please r

- Pre-req 1
- Pre-req 2
- Pre-req 3
- Pre-req 4

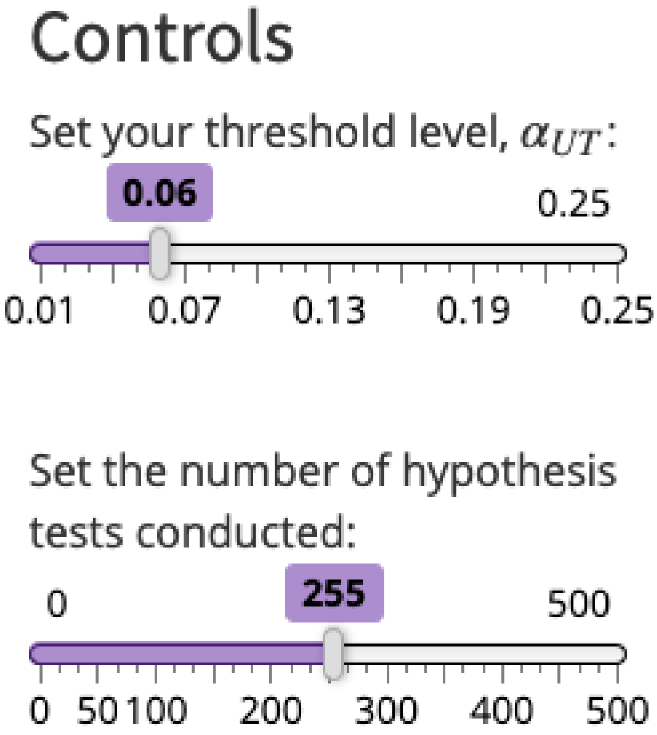
Notice the use of an unordered list; users can m

Null Hypothesis Significance Tests (NHSTs)

***p* -values**

The probability that our selected estimator tak
observed given our null hypothesis. If we were
the null hypothesis accurately modeled what w
estimator to produce a value at least as extrem
time. The larger the p-value, the more often we
least as extreme as what we've seen; the small

Figure 9.11: Collapsible Boxes Using the Purple Palette



Controls

Set your threshold level, α_{UT}:

0.06

Set the number of hypothesis tests conducted:

255

Figure 9.12: Sliders Using the Purple Palette

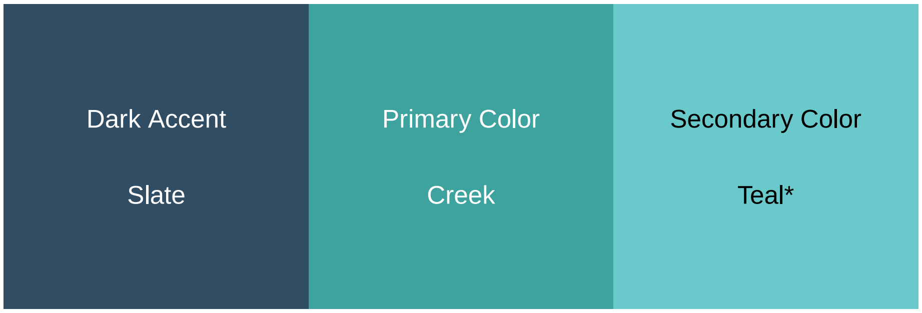
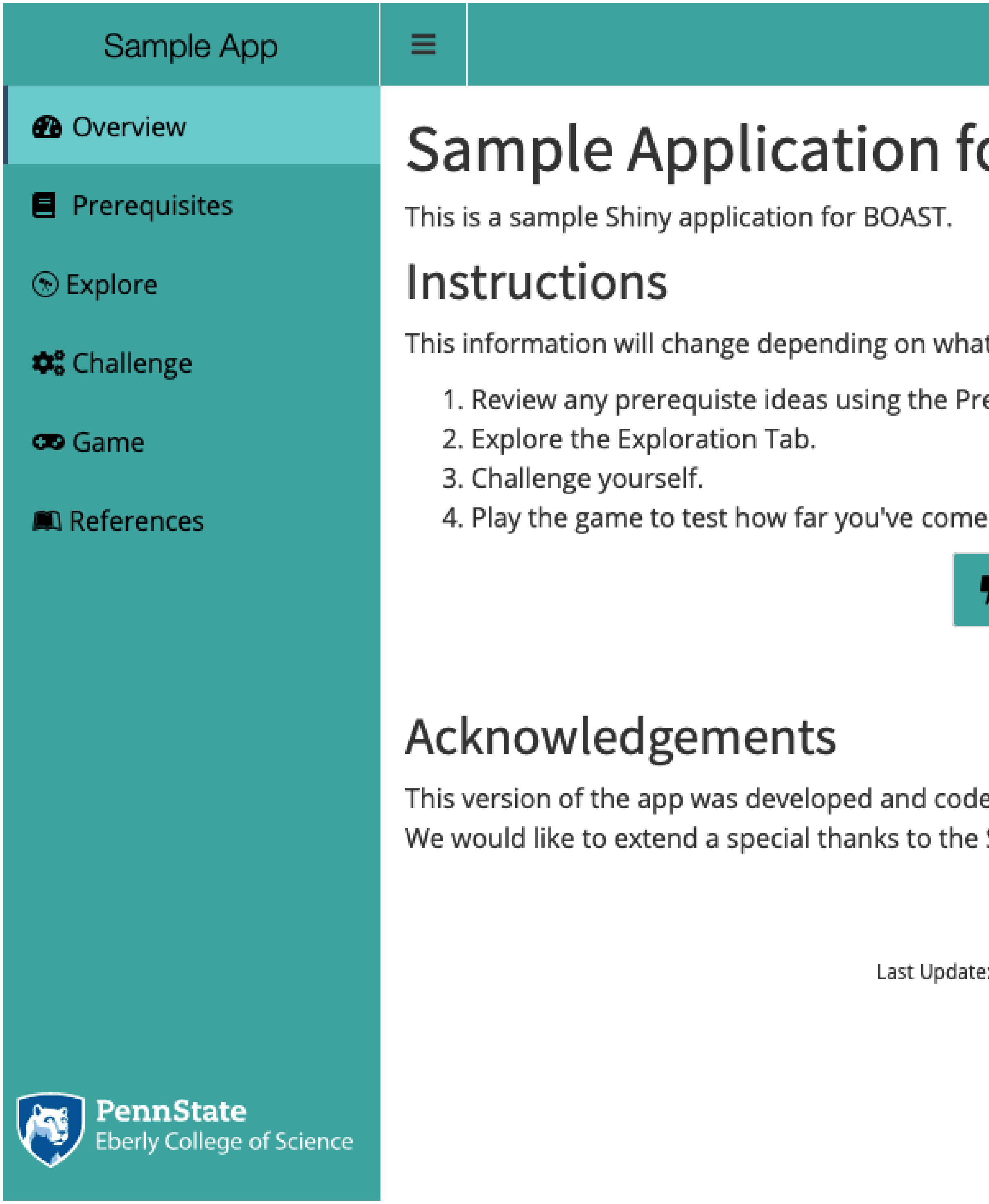


Figure 9.13: The ’Black’ Palette



Sample Application fo

This is a sample Shiny application for BOAST.

Instructions

This information will change depending on what

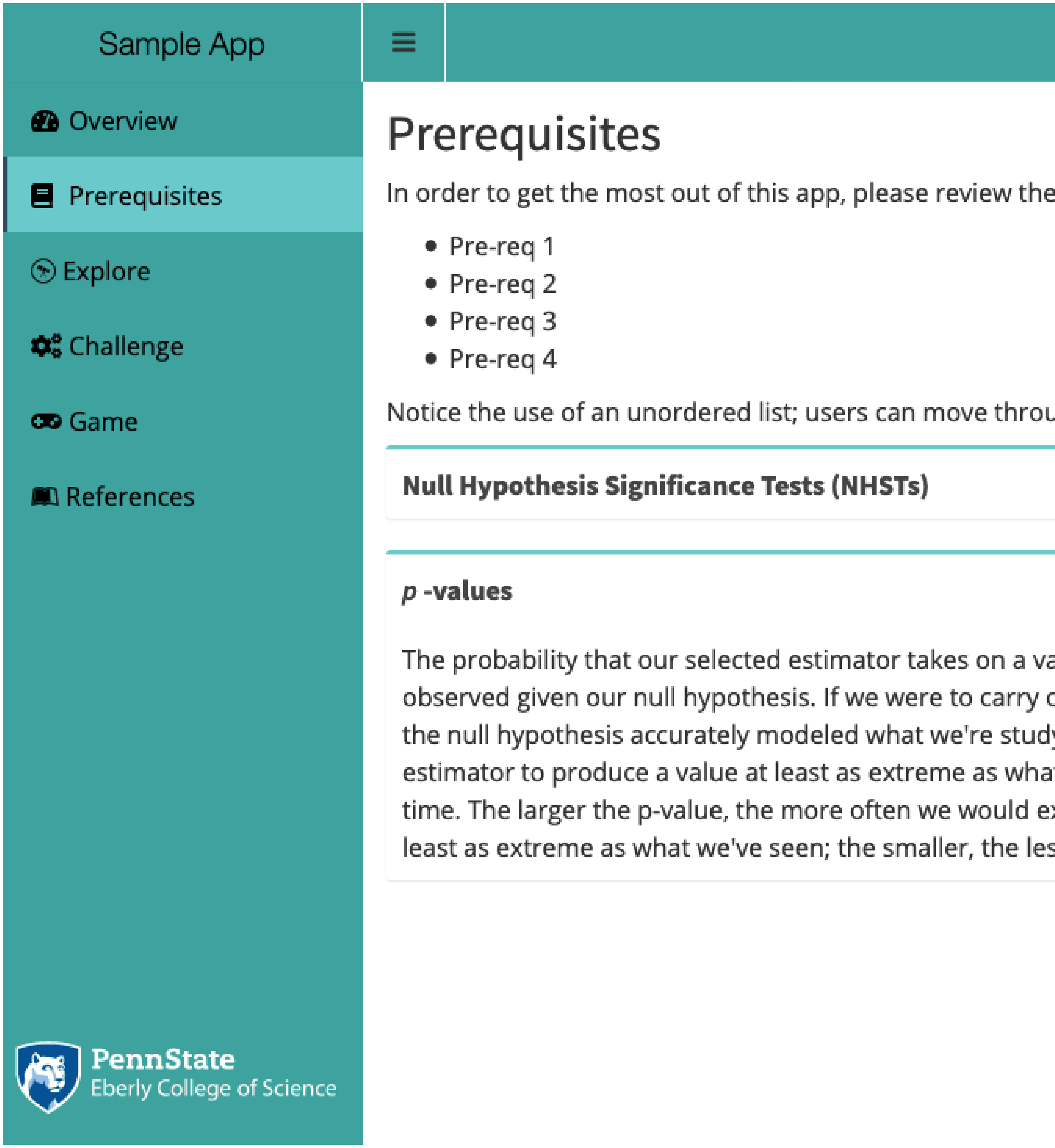
1. Review any prerequiste ideas using the Pre
2. Explore the Exploration Tab.
3. Challenge yourself.
4. Play the game to test how far you've come

Acknowledgements

This version of the app was developed and code
We would like to extend a special thanks to the

Last Update:

Figure 9.14: Overview Page Using the 'Black' Palette



Prerequisites

In order to get the most out of this app, please review the

- Pre-req 1
- Pre-req 2
- Pre-req 3
- Pre-req 4

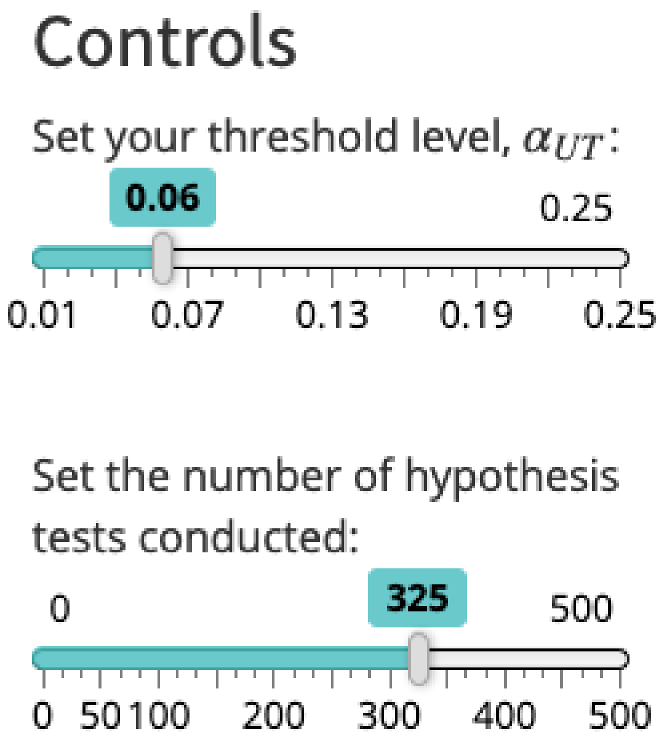
Notice the use of an unordered list; users can move throu

Null Hypothesis Significance Tests (NHSTs)

***p* -values**

The probability that our selected estimator takes on a va
observed given our null hypothesis. If we were to carry o
the null hypothesis accurately modeled what we're study
estimator to produce a value at least as extreme as wha
time. The larger the p-value, the more often we would ex
least as extreme as what we've seen; the smaller, the les

Figure 9.15: Collapsible Boxes Using the 'Black' Palette



Controls

Set your threshold level, α_{UT}:

0.06

Set the number of hypothesis tests conducted:

325

Figure 9.16: Sliders Using the 'Black' Palette

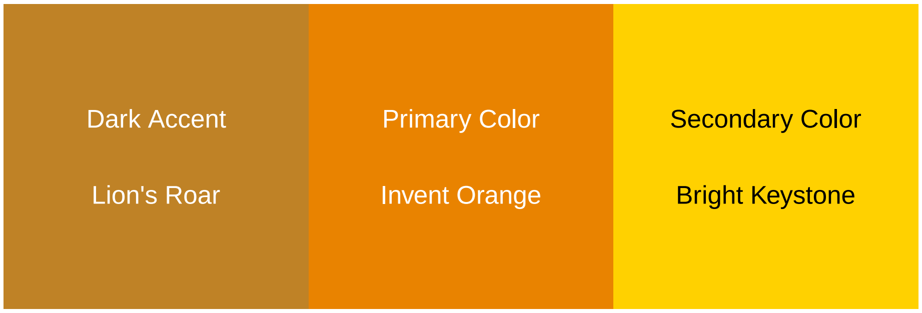
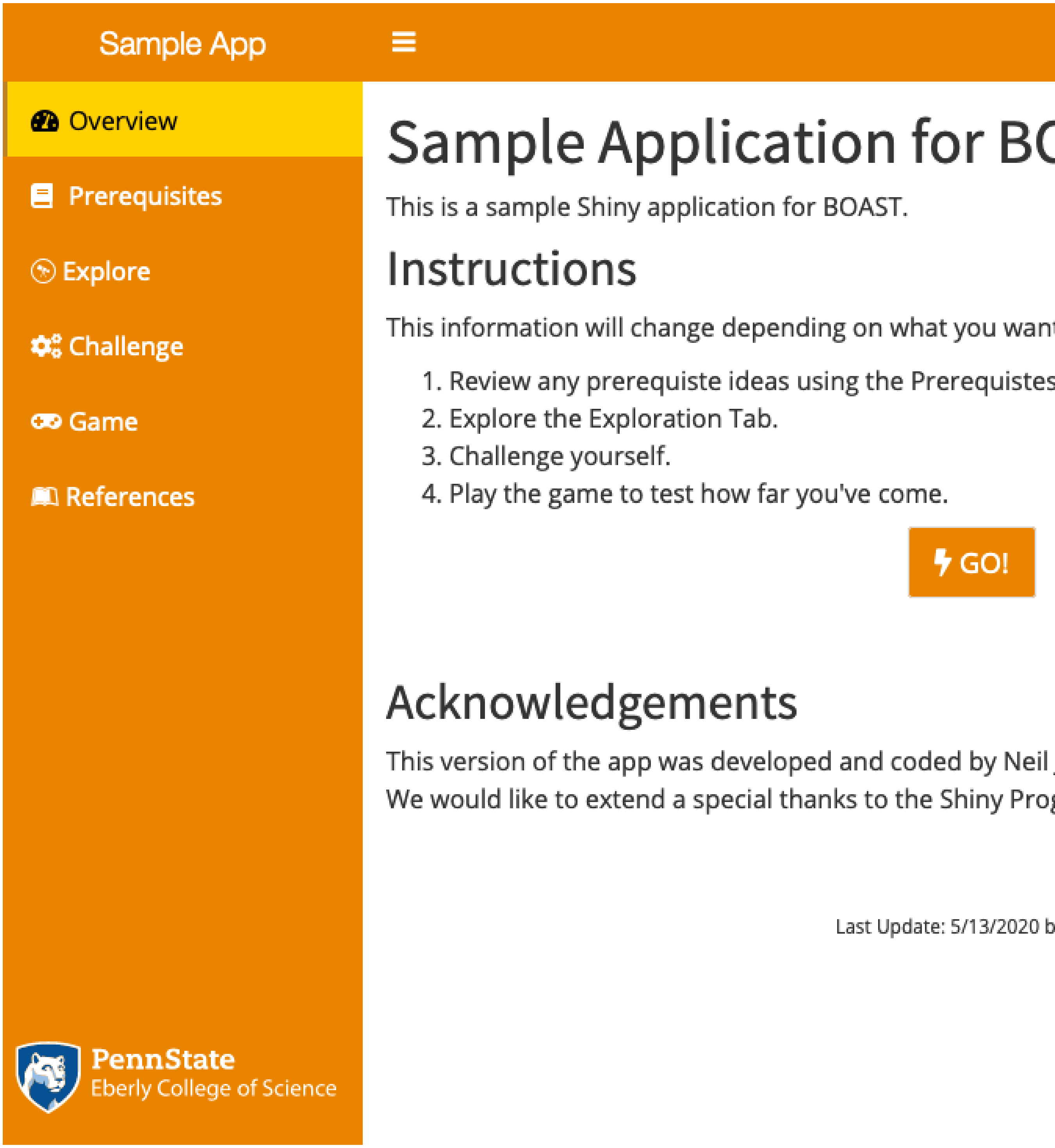


Figure 9.17: The Yellow Palette



Sample Application for BO

This is a sample Shiny application for BOAST.

Instructions

This information will change depending on what you want

1. Review any prerequiste ideas using the Prerequistes
2. Explore the Exploration Tab.
3. Challenge yourself.
4. Play the game to test how far you've come.

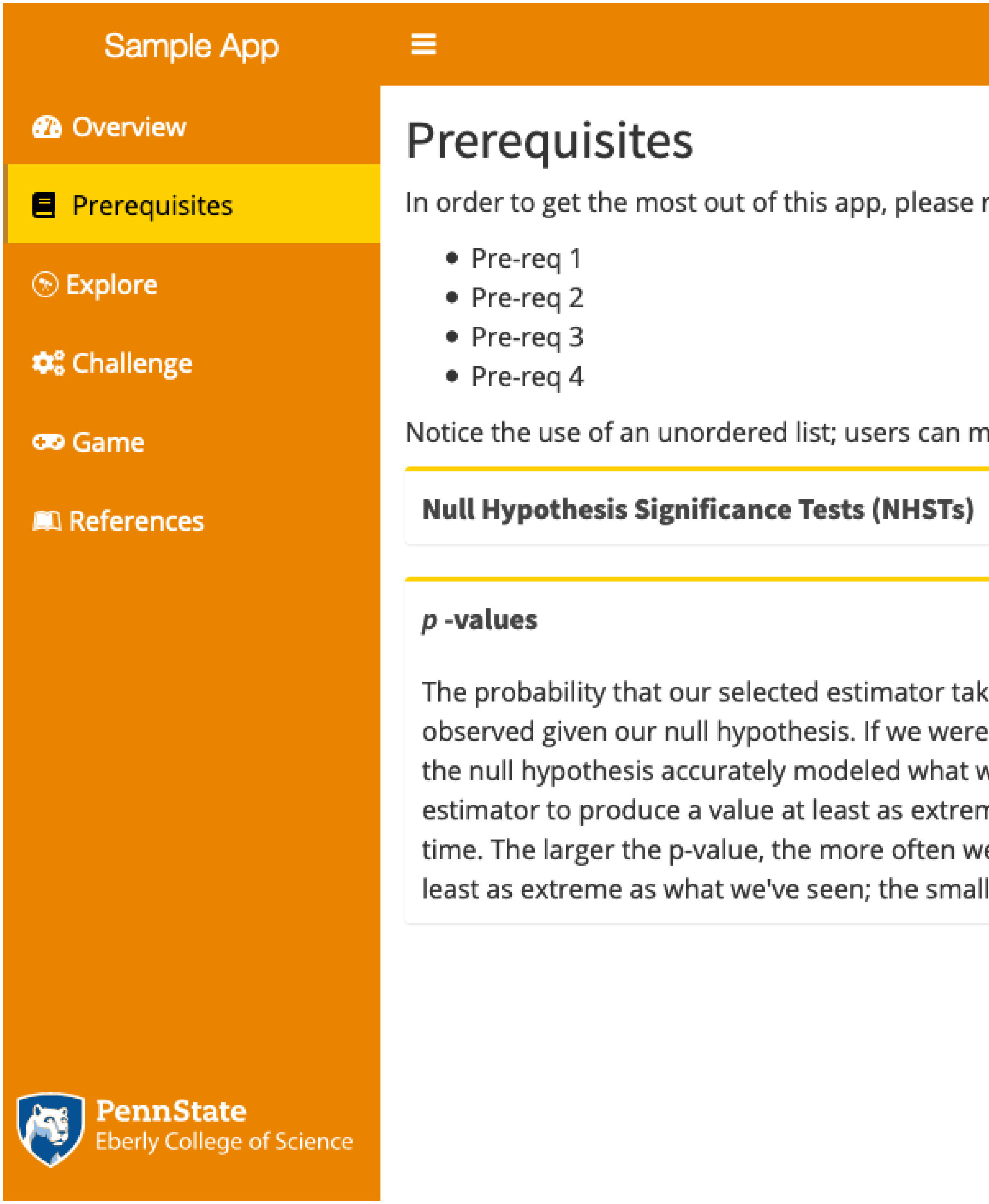
Acknowledgements

This version of the app was developed and coded by Neil

We would like to extend a special thanks to the Shiny Prog

Last Update: 5/13/2020 b

Figure 9.18: Overview Page Using the Yellow Palette



Prerequisites

In order to get the most out of this app, please r

- Pre-req 1
- Pre-req 2
- Pre-req 3
- Pre-req 4

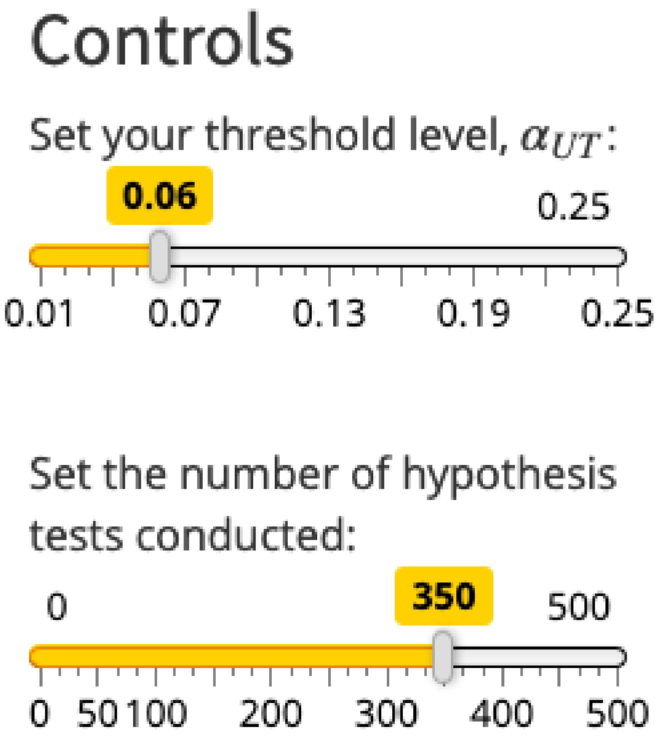
Notice the use of an unordered list; users can m

Null Hypothesis Significance Tests (NHSTs)

***p* -values**

The probability that our selected estimator tak
observed given our null hypothesis. If we were
the null hypothesis accurately modeled what w
estimator to produce a value at least as extrem
time. The larger the p-value, the more often we
least as extreme as what we've seen; the small

Figure 9.19: Collapsible Boxes Using the Yellow Palette



Controls

Set your threshold level, α_{UT}:

0.06

Set the number of hypothesis tests conducted:

350

Figure 9.20: Sliders Using the Yellow Palette

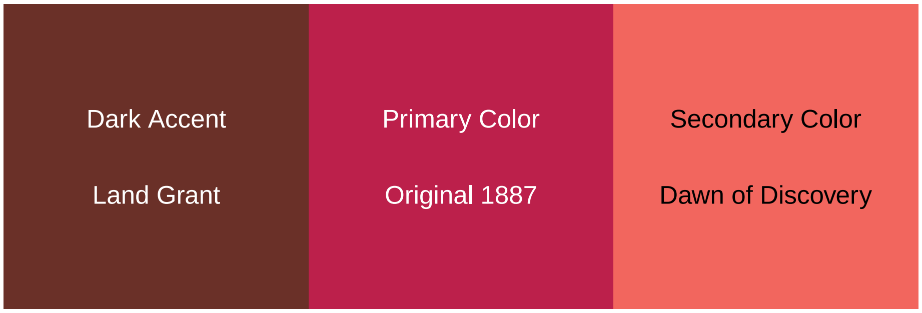
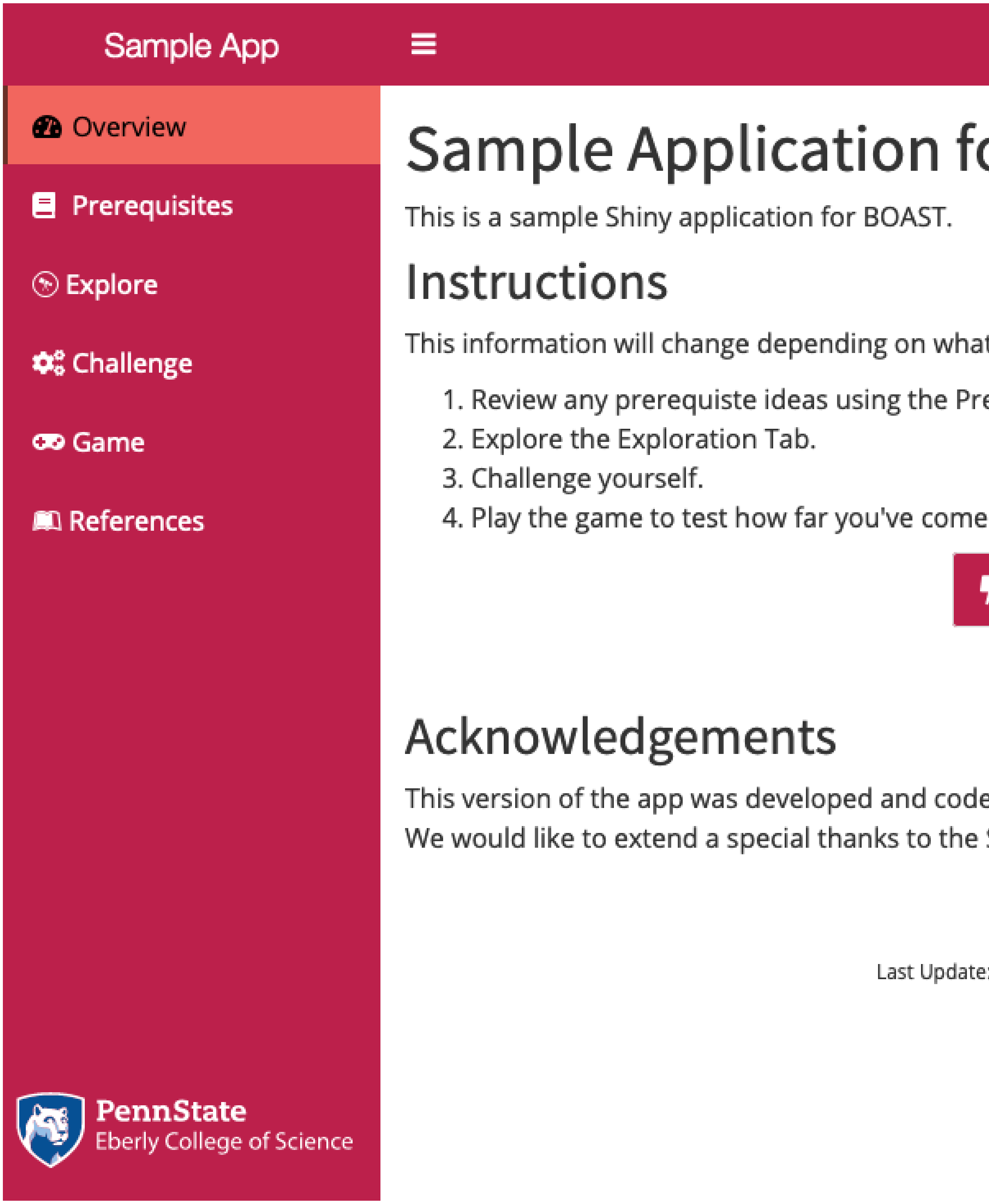


Figure 9.21: The Red Palette



Sample Application fo

This is a sample Shiny application for BOAST.

Instructions

This information will change depending on what

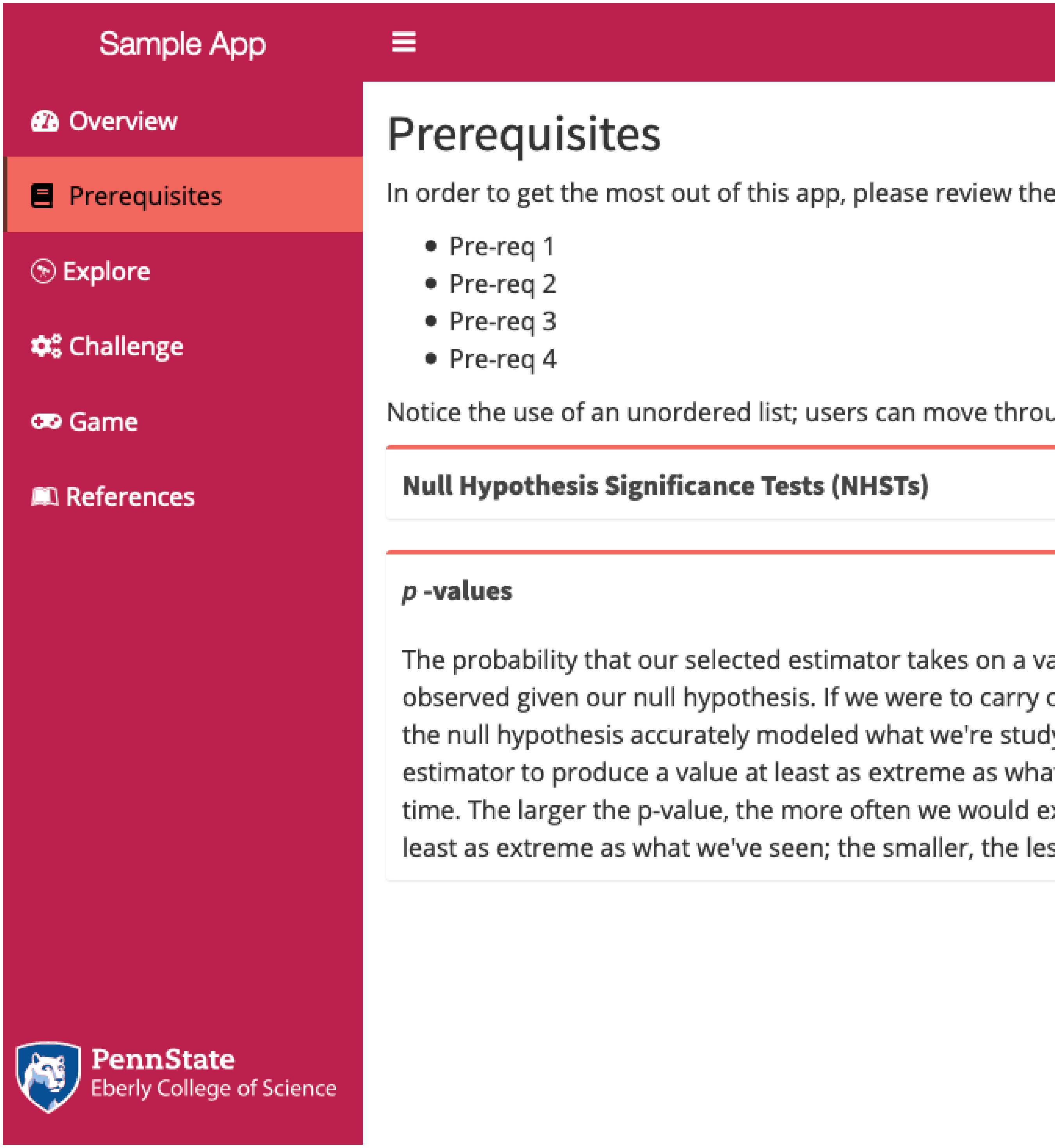
1. Review any prerequiste ideas using the Pre
2. Explore the Exploration Tab.
3. Challenge yourself.
4. Play the game to test how far you've come

Acknowledgements

This version of the app was developed and code
We would like to extend a special thanks to the S

Last Update:

Figure 9.22: Overview Page Using the Red Palette



Prerequisites

In order to get the most out of this app, please review the

- Pre-req 1
- Pre-req 2
- Pre-req 3
- Pre-req 4

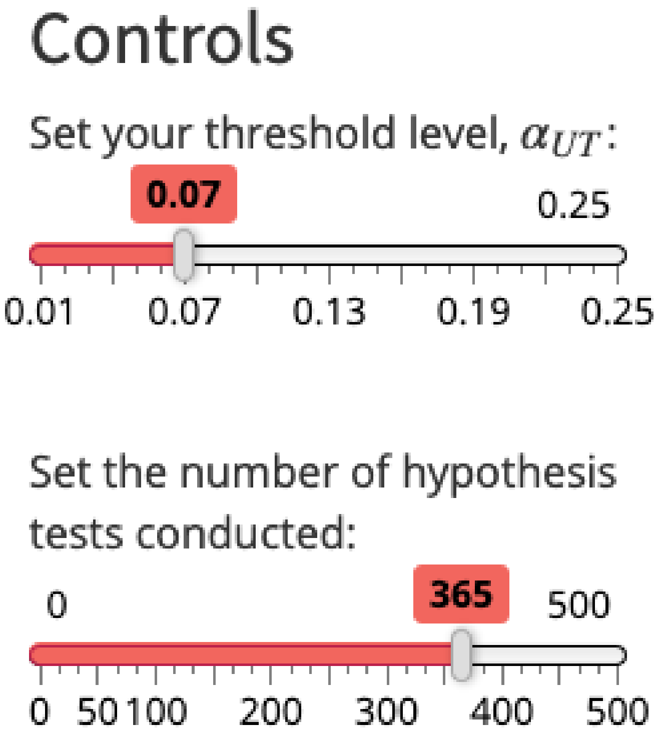
Notice the use of an unordered list; users can move throu

Null Hypothesis Significance Tests (NHSTs)

***p* -values**

The probability that our selected estimator takes on a va observed given our null hypothesis. If we were to carry o the null hypothesis accurately modeled what we're study estimator to produce a value at least as extreme as what time. The larger the p-value, the more often we would ex least as extreme as what we've seen; the smaller, the les

Figure 9.23: Collapsible Boxes Using the Red Palette



Controls

Set your threshold level, α_{UT}:

0.07

Set the number of hypothesis tests conducted:

365

Figure 9.24: Sliders Using the Red Palette

9.2 Dashboard Header

Each Dashboard Header contains only a couple of elements. The most important of these will be a [shortened] Title of your App. This will automatically be followed by the sidebar collapse/expand button. At the far right, you will then include a link to the home page of BOAST using the Home icon.

Additional icons might be included to the left of the Home icon. However, these icons remain the same for all Tabs/pages of your App and are thus are not appropriate for Tab/page specific information.

There should not be any additional elements in the Dashboard Header. Any links for navigate in your App should appear in the Sidebar on the left edge.

The width of the Title component of the Header should be 250; `titleWidth = 250`.

9.2.1 Creating the Dashboard Header

You will use the same structure regardless if you are using `app.R` or `ui.R`.

```
# [omitted code]
dashboardHeader(
      title = "Sample App", # You may use a shortened form of the title here
      titleWidth = "250",
      # the following is OPTIONAL
      tags$li(class = "dropdown", actionLink("info", icon("info"))),
      # You will see the following commented code in the Sample App;
      # you may delete this from your app
      # tags$li(class = "dropdown",
      #         tags$a(href='https://github.com/EducationShinyAppTeam/BOAST',
      #                icon("github"))),
      # the following is REQUIRED and must be last.
      tags$li(class = "dropdown",
              tags$a(href='https://shinyapps.science.psu.edu/', icon("home")))
)
# [omitted code]
```

A couple of things to notice:

- The `dashboardHeader` acts as a list environment, so it is safe to use `tags$li` here even though there isn't a `tags$ol` or `tags$ul`.
- There are not a lot of elements to the Header. We only add additional elements here if it is something static (i.e., un-changing) that is necessary for all pages of your App.
- You will need to set `class = "dropdown"` for each element after the `title` and `titleWidth`.
- The `info` icon is optional; if you use this keep in mind two things:
  - You will need to define what the link does in the `server` definition.

    - This link's action remains the same for all tabs of your app. Thus, you should not use this to store or be the main conveyance of critical information/instructions for using a particular tab of your App.
- The last element of the Header (i.e., the rightmost) will always be the home icon which takes the user to the BOAST home page.

9.2.2 What's Needed in the Server Definition

If you just have a bare bones Header (i.e., title and home button), you do not need to add anything special to your server definition.

If you do have something, for example, an Info button. You will need to add some additional code to your server definition. The code you add will depend upon the element. In general, for alert messages, we recommend that you use `shinyWidgets::sendSweetAlert`. Here's a generic example of what could be used for the Info button.

```r
# Required Packages
library(shinyWidgets)

# Move to your server definition
# Either look for server <- function(input, output, session) or open the server.R file

# [omitted code]

observeEvent(input$info, { # Replace "info" with the appropriate id
  shinyWidgets::sendSweetAlert(
    session = session, # This should stay as is
    title = "App Information",
    text = paste("[Message you want to give to the student]",
                 "Use paste with multiple lines to",
                 "improve code reability."),
    type = "info" # This option will depend upon the nature of your message
  )
})
```

The `type` argument will take one of several options. use the one that best aligns with your purposes.

- `info`: If you are just conveying some general information to your user, you'll use this value. This is the value that we will use in the vast majority of cases.
- `error`: Use this if your message tells the user that have committed some action that causes your App to fail.
- `question`: Use with an alert where you are asking the user to respond to a question.
- `warning`: If you want to give your user the opportunity to stop from doing something destructive (e.g., deleting data values), this would be

appropriate.

- `success`: Use this if your message let's the user know that something worked correctly.

Note: these buttons/links will throw an error when using the WAVE tool. This is expected at this time.

9.3 Dashboard Sidebar

The Sidebar is the main navigational tool for your users. Thus, if you want your user to access a certain tab/page, you must be sure to include this in the Sidebar. The only exception is if you are creating a series of levels inside the same Exploration, Challenge, or Game. These are referred to as Tabs Inside the Body and are covered in their own section.

The Sidebar should have a width of 250, (`width = 250`).

For full descriptions of each type of Tab, please refer to the Dashboard Body Section.

9.3.1 Creating the Dashboard Sidebar

This code will be the same regardless if you use `app.R` or `ui.R`.

```r
# [omitted code]
dashboardSidebar(
  width = 250,
  sidebarMenu(
    id = "tabs",
    # Overivew is REQUIRED
    menuItem("Overview", tabName = "Overview", icon = icon("dashboard")),
    # Prerequisites is optional
    menuItem("Prerequisites", tabName = "Prerequisites", icon = icon("book")),
    # At least one of the next three is REQUIRED
    menuItem("Explore", tabName = "Explore", icon = icon("wpexplorer")),
    menuItem("Challenge", tabName = "Challenge", icon = icon("gears")),
    menuItem("Game", tabName = "Game", icon = icon("gamepad")),
    # References is REQUIRED
    menuItem("References", tabName = "References", icon = icon("leanpub"))
  ),
  # PSU Logo is REQUIRED and to be last
  tags$div(
    class = "sidebar-logo",
    boastUtils::psu_eberly_logo("reversed")
  )
)
# [omitted code]
```

Notice that after setting the `width` of the Sidebar, you will need to call the `sidebarMenu` function. This will create the appropriate structure for your Sidebar.

9.3.2 Sidebar Order

To ensure consistency across all apps, the Sidebar needs to have the following order:

1. The Overview Tab should always come first.
2. If used, a Prerequisites Tab will come second.
3. An Activity Tab (Explore, Challenge, or Game) will come next.
4. If there are multiple Activity Tabs, then
    a. Order by Concept so that Tabs dealing with the same idea are together,
    b. Then order by Explore, Challenge, then Game
    c. If a Game Tab covers multiple Explore/Challenge Tabs, place after the last of the Explore/Challenge Tabs
    d. For a set of Activity Tabs of the same type (e.g., three Explore Tabs), the order will be up to you and the learning goals for your App
5. The References Tab will always come after the Activity Tabs and be the last element of the `sidebarMenu`.
6. The PSU Logo will always be the last element of the Sidebar, and outside of the `sidebarMenu` call. Please refer to the Section 10.1.

Note: this order will also dictate how you should organize your code in the Dashboard Body.

9.3.3 Sidebar Names

There are three Tabs whose names are fixed (i.e., you can not change): the Overview, Prerequisites, and References. The Activity Tabs will be named in the following manner:

- If there is only one Activity Tab of each type, you may use the names Explore, Challenge, and Game.
- If there are multiple Tabs of each type, you will need to rename each tab as appropriate. For example,
    - In the NHST Caveats App, there are three Explore tabs: the Multiple Testing Caution, the Large Sample Caution, and the Small Sample Caution.
    - In the One-way ANOVA App there are two Games: a Matching Game and a Fill in the Blank game.

9.3.4 Sidebar Icons

Each type of Tab that appears in the Sidebar must use a specific icon:

- Overview – tachometer-alt (formerly called dashboard),
- Prerequisites – book,
- Explore Tabs – wpexplorer,
- Challenge Tabs – cogs (formerly called gears),

- Game Tabs – gamepad,
- References – leanpub,

If you come across any of these types of tabs that have a different icon or a missing icon, please create an issue in GitHub and/or fix.

If there is a type of tab that does fit these, please talk to Neil to see about what we need to add.

9.3.5 Submenus

Given the nature of our Apps, there is **NO** reason for having submenus. If you come across an app that has submenus or you believe that a submenu is necessary, then that is a good sign that you are looking at a "bloated" app. These apps need to be marked for review to investigate breaking the app in to two or more apps.

9.3.6 What's Needed in the Server Definition

If you have followed the above specifications, you do not need to add anything special to your server definition.

9.4 Dashboard Body

The Dashboard Body is where all content (text, images, plots, buttons, etc.) exists for the user to read, view, and interact with. Thus, this is the most important part of the layout of your App.

The order in which your code the Tabs in the Dashboard body needs to mirror the order of the tabs in the Sidebar. Thus, the first `tabItem` in the `dashboardBody` should be the Overview; the last, the References.

The Dashboard Body will begin with the following code in the UI section:

```
# [code omitted]
dashboardBody(
  tabItems(
    tabItem(
      tabName = "Overview", # needs to match the names you used in the Sidebar
      withMathJax(), # if you need to display mathematics, include this line
      # [code for the tab]
    ),
    # repeat tabItem chunk for each subsequent tab
  )
)
# [code omitted]
```

The following subsections explain the purposes of each type of Tab.

9.4.1 The Overview Tab

This Tab is **REQUIRED** for all Apps. This is the main landing page of your App and should appear at the top of the Sidebar. The icon for this Tab must be "dashboard".

The Overview Tab must contain **ALL** of the following elements:

1. "Long/Formal App Title" (as Heading 1; this will be the **only** instance of Heading 1 in your App)
2. A description of the app (as paragraph text under the title)
3. "Instructions" (as Heading 2)
4. General instructions for using the App (using an Ordered List environment)
5. A button that will take the user to the next Tab/page (see Section 8.5.2 on buttons)
6. "Acknowledgements" (as Heading 2)
7. A listing of acknowledgements including, coders, content writers, etc. (as a paragraph)
8. Last Element: `div(class = "updated", "Last Update: mm/dd/yyyy by FL.")` with mm/dd/yyyy replaced with the date of the update you pushed to the server and FL replaced with your initials.

The purpose of the Overview Tab is the act like the front/home page of any newspaper, magazine, or website. Set the stage for what the user will be doing.

9.4.1.1 Creating the Overview Tab

Here's an example of making an Overview Tab

```
# Required Package
library(shinyBS)

# In the UI Section
# Inside the Dashboard Body

tabItem(
  tabName = "Overview",
  withMathJax(),
  h1("Sample Application for BOAST Apps"), # This should be the full name.
  p("This is a sample Shiny application for BOAST."),
  p("While not a proper app for helping students learn some statistical concept,
    this app functions as an example for a variety of features."),
  h2("Instructions"),
  p("This information will change depending on what you want to do."),
  tags$ol(
    tags$li("Review any prerequiste ideas using the Prerequistes tab."),
    tags$li("Explore the Exploration Tab."),
    tags$li("Challenge yourself."),
    tags$li("Play the game to test how far you've come.")
    ),
  ##### Go Button--location and text will depend on your goals
  div(
    style = "text-align: center",
    bsButton(
      inputId = "explore1",
      label = "Explore!", # Notice there are NO spaces between the letters
      size = "large",
      icon = icon("bolt"),
      style = "default"
      )
    ),
  ##### Create two lines of space
  br(),
  br(),
  h2("Acknowledgements"),
  p("This version of the app was developed and coded by Neil J. Hatfield and Robert P.
    Carey, III.",
    br(),
    "We would like to extend a special thanks to the Shiny Program Students.",
    #### Create three lines of space
    br(),
    br(),
    br(),
    div(class = "updated", "Last Update: 5/13/2020 by NJH.")
  )
)
```

A few things to notice:

- If you need a new line but not a new paragraph, you use the `br()` tag.
- The label for the button needs to be in-line with Section 8.5.2.
- There should **NOT** be any spaces between letters of a button label. This is a violation of Accessibility as this destroys the label. While we might read "G (space) O" as the word "go", a screen reader reads out "gee" (pause) "oh" to the user.
- There is no need to use boldface or colons with the section headings when you properly use Heading tags. Thus, "Instructions:" does not follow this

Style Guide.
- There should not be an “About” heading. The text between the Title of your App and the Instructions head serves as the description.

9.4.1.2 What’s Needed in the Server Defintion

At bare minimum you will need to have one element in your server definition: the action for your button. If you have multiple buttons, you might need to have several more code chunks.

Here is a generic example for the button on the Overview Tab that moves the user to the appropriate next Tab:

```
## Define what each button does; repeat this style of coding for each button

# In your server section
# [code omitted]
observeEvent(
  eventExpr = input$expore1, #append the button's inputId to input$ as the event expre
  handlerExpr = { # This is the action portion of your button and must be in { }
  updateTabItems(session, # This how you allow the user to move Tabs
                 inputId = "tabs", # the id of your Sidebar
                 selected = "Explore" # Name of Tab to go to
                 )
})
```

In the rare cases where you’re having a button in the Overview Tab do something other than move to a new tab, you’ll change the `handlerExpr` to do that other action.

9.4.1.3 Buttons that Go to Activity Packets

An exception to the above is if the button is to allow the user to download/open up an activity file. In these cases, you’ll not place any code in the server definition. Rather, you’ll change the nature of the button. Specifically, you’ll not use `bsButton` but `actionButton`:

```
# In the UI Section
actionButton(
  inputId = "ap1",
  label = "Activity Packet",
  icon = icon("cloud-download"),
  onclick = "window.open('../../ActivityPackets/Caveats/')"
)
```

Notice that the `onclick` argument is what creates the action for button and references a relative path to a particular file. In this case a R Markdown file that lives the in directory called. (The actual file name isn’t listed to ensure that the processed/rendered version of the RMD is what the users see.)

9.4.2 A Prerequisites Tab

If your App needs to ensure that the user has the base understandings necessary to interact with your App, you'll need to create a prerequisites Tab. Otherwise, skip this Tab.

The icon for this Tab must be "book".

Use the word "Prerequisites" rather than "Pre-reqs", "Prereqs", or "Pre-requisites".

9.4.2.1 Types of Prerequisites

There are two different types of prerequisites: technical/conceptual and contextual. Both of these go into the Prerequisites tab.

Technical/Conceptual Prerequisites cover ideas that the user needs in order to fully engage with your App's statistical goal. For instance, if your App is about ANCOVA, the ideas of ANOVA and building a linear model would be good candidates for technical/conceptual prerequisites.

Contextual Prerequisites cover ideas that which are beneficial for the user to understand a context you're using. For example, if you are referencing an astragalus, you should include a brief explanation and/or picture of an astragalus.

Keep in mind that Contextual Prerequisites are different than context which should be part of the Activity Tab. If the information is necessary to interpret sliders/graphs and is *specific*, then you should include this information in the Activity Tab. If the information helps the user say "Oh, that's what they mean by [blank]", that is good sign of something to put in the Prerequisites Tab.

9.4.2.2 Text Links in Prerequisites (and Beyond)

In as many instances as possible, we would like to provide the user with a link to Online Notes of a World Campus Statistics course.

Note: what appears here is applicable any time you want to link to an webpage that is beyond BOAST.

The link that you provide must take the user to the appropriate location. Do not send the user to the home page for a course; rather, take them to the relevant page. To do this, you'll need to explore the Department of Statistics STAT ONLINE page and look through the courses.

You will create these links in-line, not as a button. Thus, they must be part of a paragraph block (i.e., inside a `p()` with other text) or as part of list item (i.e., inside a `li()`).

Your link must include descriptive text. Using "Click Here" is **not** descriptive. Rather say where the link will take the user. If you look through the links that we've included in this Style Guide, we've been modeling this. This descriptive text not only helps all users anticipate where they are going but also improves the accessibility of the links. (Plus, have you ever tried to click a small link on your phone?)

Once you find the appropriate page, you'll need to copy the URL for your link. There are some instances where we might be able to find an existing anchor (look for two inter-locking rings to appear when you place your cursor over a title) or make a request for adding an anchor. These are especially useful if what you want to link to is only part of the page.

*Note: not all requests for anchors may be fulfilled and not all course notes have anchors.

The styling of the link will be managed by the BOAST CSS file.

Here's are two examples of how you would code a text link:

```
# [omitted code]
# Working in the UI section

# Example 1: in a paragraph
p("While not critical, you might wish to refresh your understanding on some of the basi
  of graphs in statistics. A good resource for this would be the ", # Notice the ending
  tags$a(
    herf = "https://online.stat.psu.edu/stat100/lesson/3/3.2#graphshapes", #the URL
    "STAT 100 Table of Graph Shapes" # the descriptive text for the link
    ),
  ". Feel free to check that resource out."
  # Notice the ending punctuation for the prior sentence is not part of the link.
)

# Example 2: in a list item
tags$ul(
  tags$li("Review the ", # Notice the ending space
    tags$a(
      herf = "https://online.stat.psu.edu/stat100/lesson/3/3.2#graphshapes", #the URL
      "STAT 100 Table of Graph Shapes" # the descriptive text for the link
      )
    # List items don't necessarily need ending punctuation.
    #Be consistent; either all items do or none.
  )
)
# [omitted code]
```

9.4.2.3 Creating a Prerequisites Tab

Here's an example of the code needed to create the Prerequisites Tab in the UI:

```
# [code omitted]
tabItem(
  tabName = "Prerequisites",
  withMathJax(), # this line only need if you display mathematics
  h2("Prerequisites"),
  p("In order to get the most out of this app, please review the following:"),
  tags$ul(
    tags$li("Pre-req 1"),
    tags$li("Pre-req 2"),
    tags$li("Pre-req 3"),
    tags$li("Pre-req 4")
    ),
    p("Notice the use of an unordered list; users can move through the list any way they wish."),
  p("A second style of doing prerequisites is with collapsible boxes:"),
  box(
    title = strong("Null Hypothesis Significance Tests (NHSTs)"),
    status = "primary",
    collapsible = TRUE,
    collapsed = TRUE,
    width = '100%',
    "In the Confirmatory Data Analysis tradition, null hypothesis significance tests serve as a
    critical tool to confirm that a particular theoretical model describes our data and to make a
    generalization from our sample to the broader population (i.e., make an inference). The null
    hypothesis often reflects the simpler of two models (e.g., 'no statistical difference',
    'there is an additive difference of 1', etc.) that we will use to build a sampling
    distribution for our chosen estimator. These methods let us test whether our sample data are
    consistent with this simple model (null hypothesis)."
  ),
  box(
    title = strong(tags$em("p"), "-values"),
    status = "primary",
    collapsible = TRUE,
    collapsed = FALSE,
    width = '100%',
    "The probability that our selected estimator takes on a value at least as extreme as what we
    observed given our null hypothesis. If we were to carry out our study infinitely many times
    and the null hypothesis accurately modeled what we're studying, then we would anticipate our
    estimator to produce a value at least as extreme as what we have seen 100*(p-value)% of the
    time. The larger the p-value, the more often we would expect our estimator to take on a value
    at least as extreme as what we've seen; the smaller, the less often."
  )
),
# [code omitted]
```

For more information on collapsible boxes, see Section 8.5.6.

9.4.2.4 What's Needed in the Server Definition

Generally speaking, the purpose of the Prerequisites Tab is to convey key background information for the user to double check they understand before moving into the heart of your App. Thus, this tab contains static text and images. You do no need to have anything in the server definition for the Prerequisites Tab.

9.4.3 Activity Tab(s)

The heart of your App is the one or more tabs where users interact with the App beyond simple navigation. These are the Activity Tabs. Some apps will have a single activity, others several; deciding on how many is part of the design process.

Currently, we have three types of Activity Tabs in BOAST:

- Exploration/Explore Tabs,
  - These tabs center around the user exploring the target concept.
  - These tabs will generally have more text on the page here than other types of Activity Tabs.
  - There are often guiding questions meant to help the user engage in productive explorations.
  - The goal is not to assess the user's understanding, but to support their construction of productive meanings for the concept.
- Challenge Tabs,
  - These tabs center around a user challenging themselves by testing out their understanding of a concept.
  - While there might still be a fair amount of text on the page, there will be less than an Exploration Tab.
  - Questions here will be in-between a guiding question and an assessment question.
  - The goal is to provide the user an opportunity to test and refine their understandings.
- Game Tabs,
  - These tabs center around the user review a concept (or several) in a game like format.
  - These generally have the least amount of non-question text on the page (i.e., instructions).
  - The goal is to provide an opportunity for a student to review and practice one or more concepts.

9.4.3.1 General Layout for Activity Tabs

While there are some differences between the different types of Activity Tabs, there is one firm constant for all of them:

Each Tab should contain all information/instructions for the user to be able to interact with the activity without having to switch to other

Tabs.

There is nothing worse for the user than getting to an Activity Tab and not knowing what they are supposed to do. The Instructions on the Overview are for using the *entire* App, not any one particular page. Thus, you need to have specific page instructions somewhere on the page.

Keep in mind that this is beneficial to all users, but especially those who are using assistive technology such as screen readers. If you put your instructions on a separate tab, you are now requiring that your user memorize those instructions. This is already cognitively demanding for sighted individuals, but for users with vision impairments, even more so.

In addition, we will adopt a 3-part layout:

- Across the top will be any general information the user needs to interact with your App as well as an context information.
- To the left and wrapped in a well panel will be the inputs/controls that the user will need to manipulate.
- To the right and NOT in a well panel will be the outputs (graphs, images, R output)

There will be some cases where this general layout does not necessarily work. For instance, Tic-Tac-Toe games will not follow this layout.

9.4.3.2 UI and Server Definitions

At this time, we do not have any examples of creating these tabs for this Style Guide. Rather, we encourage you to look at the many examples in the Book of Apps for Statistics Teaching as well as the GitHub Repository for BOAST.

Keep in mind that of all tabs in your App, these tabs will demand the most for the UI and the Server.

9.4.4 References

The last Tab will be for your references. This Tab is **REQUIRED** and is where you will place a reference list for all of the following items that you used in your app:

- All `R` packages you used
- Sources of any Code you used directly or drew heavily upon from other people
- Pictures and/or other images
- Data sets
- Refer to the Chapter 14 on Documentation of this Style Guide for more information.

The icon for this Tab must be “leanpub”, .

We will additionally place licensing information for your App at the bottom of the Reference page. We have created a function in `boastUtils` (version 0.1.6.1) that will automatically put the correct information on the page.

9.4.4.1 Creating a References Tab

Creating a References Tab mimics both the Overview and Prerequisites Tab structure and is done in the UI section:

```r
# In the UI Section
# [code omitted]
tabItem(
  tabName = "References",
  withMathJax(), # Rarely, if ever, will you need MathJax in the references
  h2("References"),
  p( # Each reference is in its own paragraph
    class = "hangingindent", # you must set this class argument
    "Bailey, E. (2015), shinyBS: Twitter bootstrap components for shiny, R package. Ava
    from https://CRAN.R-project.org/package=shinyBS"
    ),
  p(
    class = "hangingindent",
    "Carey, R. (2019), boastUtils: BOAST Utilities, R Package. Available from
    https://github.com/EducationShinyAppTeam/boastUtils"
    ),
  p(
    class = "hangingindent",
    "Chang, W. and Borges Ribeio, B. (2018), shinydashboard: Create dashboards with 'Sh
    Package. Available from https://CRAN.R-project.org/package=shinydashboard"
    ),
  p(
    class = "hangingindent",
    "Chang, W., Cheng, J., Allaire, J., Xie, Y., and McPherson, J. (2019),  shiny: Web
    application framework for R, R Package. Available from
    https://CRAN.R-project.org/package=shiny"
  ),
  p(
    class = "hangingindent",
    "Hatfield, N. J. (2019), Caveats of NHST, Shiny Web App. Available from
    https://github.com/EducationShinyAppTeam/Significance_Testing_Caveats/tree/Pedagogi
  ),
  p(
    class = "hangingindent",
    "Wickham, H. (2016), ggplot2: Elegant graphics for data analysis, R Package, New Yo
    Springer-Verlag. Available from https://ggplot2.tidyverse.org"
  ),
  br(), # Three blank spaces
  br(),
  br(),
  boastUtils::copyrightInfo()
)
# [code omitted]
```

9.4.4.2 What's Needed in the Server Definition

You do not need to place anything in the server definition for the References Tab.

9.4.5 Tabs Inside the Body

There are two types of tabs in a Shiny app: there are the `tabItem` (i.e., the pages within an app and should appear in the Sidebar) and `tabPanel` (i.e., creating sub-pages or independent sections). In this section, we will discuss this later case.

Deciding on whether to use `tabPanel` is going to depend on several things:

1. Do you have two or more aspects that are related enough that they shouldn't be their own separate tabs/pages of your App?
   a. If NO, then you shouldn't use `tabPanel`.

   b. If YES, then continue.
2. Are any of your aspects something that would be better suited as a Challenge or Game tab?
   a. If YES, move that aspect to a separate page. If you still have 2+ aspects, continue.

   b. If NO, continue.

3. Are the aspects independent enough that a person can skip a couple and still use the App successfully?
   a. If NO, then you should re-consider your design.

   b. If YES, then proceed with using `tabPanel` in you design.

When you go to make a set of tab panels you will need to first create a `tabsetPanel` which will wrap around all of the individual panels. Use `type = "tabs"`.

The tabs inside the body should automatically appear horizontally and along the top of the tab body (i.e., in the white space below the Dashboard Header). Any visual styling will be managed by the BOAST CSS file at a global level.

9.5 Common Elements

In addition to the Dashboard elements of the apps, there are other elements that are common. This include things such as how inputs should be ordered, buttons, correct/incorrect indicators, and animation buttons.

For information about popovers, rollovers, hover text, or tool tips, please see Section 13.2.

9.5.1 Ordering Inputs

One of the most powerful aspects of Shiny apps is that the user interacts with them. Thus, we need to consider not only the ways in which user interact (e.g., buttons, sliders, text entry, etc.) but also the order in which you want the user to manipulate the inputs. Coming up with a single declaration for how to order inputs in all cases is not necessarily feasible. However, we can set up a general guideline for how to make decisions on ordering your inputs.

Please use the following guidelines for determining the order of inputs in the User Interface (UI):

1. In general, if you want your user to do things in certain order, make your inputs appear in that order. For example, If you want them to pick a data set, then an unusualness threshold/significance level, what attribute to test, and then set a parameter value, then your inputs should appear in that order.
2. Make use of how we read the English language, i.e., Top-to-Bottom and Left-to-Right to provide an implicit ordering for your user.
3. If a user needs to carry out steps in particular sequence for your App to run properly, then place your inputs inside of an Ordered List environment with explicit text on what they should do. For example,
   1. Choose your data set: [dropdown]

   2. Set your unusualness threshold/significance level
      [slider]

   3. Which attribute do you want to test: [dropdown]

   4. What parameter value do you want to use: [numeric input]
4. If an input is going to reset other inputs you should either:
   a. Warn the user before hand

   b. Move the input to the top of the list

   c. Program the input to not reset other inputs, or

   d. Some combination of the above

5. If the inputs are not dynamically linked to the output (e.g., plots automatically update with a change in the input's value), then you should include a button that says "Make Plot" at the end of the inputs.

9.5.2 Buttons

Buttons are one way in which users interact with the apps. The two most common button functions that we use are `shiny::actionButton` and `shinyBS::bsButton`. Both functions share many of the same features. Two ways in which they are different is that `shinyBS::bsButton` has an additional `style` argument while `shiny::actionButton` has a `width` argument that gives you fine grain size control (`bsButton` just has a qualitative size option).

There are three key styling aspects to every button: shape/animation, color, and text & icon.

9.5.2.1 Shape/Animation

All shape aspects of buttons will be controlled by CSS. The standard shape will be rectangular (the default). Sizing will be controlled by CSS although setting `size = "large"` for the `bsButton` call may be done.

We have a number of apps where a button will change shape/size when a person hovers their cursor over it. This "animation" is to be discontinued. This is to say that buttons which change shape/size should be flagged as issues and resolved at the first opportunity.

At most, the button's color might change (i.e., lighten or darken only), depending on the context.

9.5.2.2 Color

The coloring of the button will also be controlled by CSS in one of two ways.

The default way will be through the BOAST CSS. This will ensure that the selected color scheme for your App will be consistent.

The second way only applies to `bsButton` and the `style` argument. Here, this option references an external CSS file beyond BOAST. We see these most often in Game Activity Tabs. Use the following list to guide you in choosing which style is appropriate:

- `warning`: Good for when you want the user to proceed with caution; for example a submit button in a game.
- `danger`: Good for when you want the user to think twice before clicking; for example, a reset game button.
- `success`: Good for when you want to convey that the user can proceed safely; for example, a button that advances the user through the game

- `info`: Good for when you want to give some additional information; for example, a button that triggers game instructions popping up, a button that gives a hint, or a button that might filter a question pool.

When in doubt, use the the `default` style option (or even omit this argument) for `bsButton` or use `actionButton`.

9.5.2.3 Text & Icon

The last styling element of a button is two-fold: the text that is in the button and the icon.

Here are some guidelines for text of a button:

- All buttons must have some text.
- Generally speaking, the text should be relatively brief and clear.
  - Don't use "Go to the next page" when you could use "Next"
- The text should make sense with the action of the button; for example,
  - "Reset" if the button resets something (a game, a plot, inputs)
  - "Submit" if the button triggers the app to grab and process input values
  - "Make Graph" if button causes a graph to be generated
  - "Show/Hide Graph" if a button makes a graph object appear/disappear
  - "Next" if a button moves the user along some path.
- If the button references something like a particular tab (prerequisites, exploration, etc.), the text should reflect this.
  - "Explore!" for a button that takes a user to an Exploration tab.
  - "Prerequisites" for a button that takes a user to a Prerequisites tab.
  - "Challenge Yourself!" for a button takes a user to a Challenge tab.
  - "Play!" for a button that takes a user to Game tab.
- If a button references an object like an activity packet or a download prompt the text should refer to that
  - "Activity Packet" for a button that would open up and/or download a packet for the user
  - "Download Data" for a button that would download a data file.
- Clarity is essential. If there are multiple buttons on the page, make sure that you use clear text for what button does and/or references.

Here are guidelines for the inclusion of icons in a button:

- Game buttons will NOT have any icons.
- Direction Buttons (e.g., "Next" or "Previous") will NOT have any icons. Rather make the button text "« Previous" or "Next »"
- A "Prerequisites" button will use the "book" icon,
- All other tab buttons (labels ending with "!") will use the "bolt" icon,
- A download button will use the "cloud-download-alt" icon,

9.5.3 Correct/Incorrect Marks

In games, you can give the user a visual cue as to whether they are correct or incorrect through the use of two images:

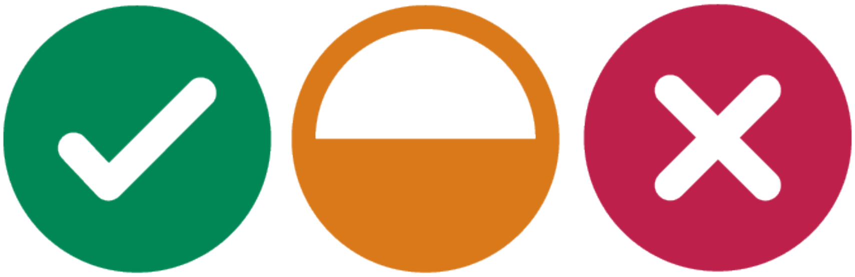


Figure 9.25: Correct, Partially Correct, and Incorrect Marks

You can save these two images by right-clicking on them and selecting "Save Image As…". You will need to put them in the `www` folder/directory of your App.

Their placement in your App will depend upon what makes the most sense.

Be sure that you add alternative text to these images; see Section 10.4.6. Do not use "right" for alt text; use "correct".

9.5.4 Animation Buttons

One feature of slider inputs is the option to include a Play/Pause button that allows the user to create an animation of your plot. Enabling this option can be quite useful if allowing the user to move through the whole set of slider values is desirable.

To enable this, you'll need to make use of the `animate` argument:

```
#[code omitted]
sliderInput(
  inputId = "mtcAlpha",
  label = "Set your threshold level, \\(\\alpha_{UT}\\):",
  min = 0.01,
  max = 0.25,
  value = 0.1,
  step = 0.01,
  animate = animationOptions(
    interval = 1000, loop = TRUE))
#[code omitted]
```

You can set `animate=TRUE`, `animate=FALSE` or invoke the `animationOptions` function as we've done in the example and recommend. This will force you to make some important decisions: namely, how long the slider should wait between each movement (`interval`, in milliseconds) and should the animation start over once the slider reaches the maximum (`loop`).

The `interval` is going to the most challenging value to figure out. This timer ***ignores*** everything else; that is, it doesn't wait to see whether your plot has updated. Remember, the more complicated the process that generates your plot is, the longer your App will need to render the plot. Thus, you can quickly get into a case where the slider has advanced several times while your App is still trying to render the first update. While `renderCachePlot` can help speed things up, keep in mind that you still might need to play around with the `interval` value to ensure smooth functionality.

Make sure when you're testing an animated slider to vary all of the parameters involved in the graph. This will help ensure that you test adequately.

The styling of the play/pause button will be controlled by the BOAST CSS file.

9.5.5 Progress Bar

Consider adding a loading bar to show the process for intense computations; this will help the user understand that your App is processing and not frozen/broken.

9.5.6 Collapsible Boxes

One technique that we can make use of to cut down on the amount of visible text on a page is through the use of collapsible boxes. Collapsible boxes are preferred than other methods in that 1) the content remains on the page and thereby accessible, and 2) the user retains control for when to show/hide this information.

Two great places to consider using collapsible boxes include the Prerequisites Tab and **static** Context Information across the top of an Activity Tab.

Note: if the context information changes (e.g., can be switched due to user actions), then collapsible boxes should ***NOT*** *be used. The context in these cases should* ***always*** *remain visible.*

9.5.6.1 Creating Collapsible Boxes

To create a Collapsible Box, you'll need to work in th UI section of your code:

```
# [code omitted]
box(
  title = strong("Title for the Box"), # Use the strong tag
  # Give either the title of the review concept or "Context"
  status = "primary", # Leave as primary
  collapsible = TRUE, # This allows collapsing
  collapsed = FALSE, # Initial value
  # If the only collapsible item, use FALSE
  # If there are multiple, the first one is FALSE, others can be set to TRUE.
  width = '100%', # use this setting
  "The text that will 'disappear' goes here."
),
# [code omitted]
```

Given the static nature of the information in a collapsible box, you should not need to add anything to the server definition.

The styling of Collapsible Boxes is controlled by the central CSS file. If you use the above code as your template, the coloring will automatically match your App's assigned color theme.

9.5.7 Well Panels

Well panels are visual styling that we use to help offset user controls (i.e., inputs) from both context and graphs (i.e., outputs).

To place something inside of a well panel, all you need to do is wrap that object in the `wellPanel` function. However, the placement of the `wellPanel` call matters.

If you wrap each individual element in `wellPanel` you'll get a separate well panel for each call. If you place `wellPanel` too high up in your code, you'll end up putting everything into the well panel.

```
# In the UI Section
# [code omitted]
# Inside a tabItem
fluidRow( # this allows you to create dynamic columns for responsive design (mobile friendly)
  column( # this is one of those columns
    width = 4, # the initial grid width; all columns must add to 12
    wellPanel( # Placing the wellPanel call here will wrap around all controls
      h3("Controls"),
      sliderInput(
        inputId = "mtcAlpha",
        label = "Set your threshold level, \\(\\alpha_{UT}\\):",
        min = 0.01,
        max = 0.25,
        value = 0.1,
        step = 0.01,
        animate = animationOptions(interval = 1000, loop = TRUE)
        ),
      br(), # creating vertical space between sliders
      sliderInput(
        inputId = "mtcTests",
        label = "Set the number of hypothesis tests conducted:",
        min = 0,
        max = 500,
        value = 5,
        step = 5
        )
      ) # this closes the wellPanel
    ), # this closes the first column
  column(
    width = 8,
    h3("Plot"),
    plotOutput("pplotMTC"),
    bsPopover(
      id = "pplotMTC",
      title = "Investigate!",
      content = "What happens to the number of statistically significant tests when you
      increase the number of tests?",
      placement = "top"
      )
    ) # closes second column
  ), #closes the fluid row
```

Part V

Style Guide-Visual Style

Chapter 10

Design Style

Design Style is the second side of the Visual Appearance of every app. Here, we deal with visual aspects that go beyond the layout of your App. This includes issues of branding, color, how you make text look (font, size, emphasis), and graphics (plots, tables, and images).

10.1 PSU Branding

Given that we are all associated with Pennsylvania State University, we need to include the Penn State logo in each App. Rather than sticking the logo at the top of the Overview page, we are going to place the logo at the bottom of the sidebar. This has the benefit of having the logo appear throughout the entire App AND making the logo be as unobtrusive as possible.

In your UI section of `app.R` (or the `ui.R` file), at the end of the `dashboardSidebar()` section, you will need to include the following:

```
tags$div(class = "sidebar-logo",
         boastUtils::psu_eberly_logo("reversed"))
```

Here's how this code would look in context:

```
dashboardPage(
  skin = "blue",
  dashboardHeader(
    title = "title",
    # [omitted code]
  ),
  dashboardSidebar(
    width = 300,
    sidebarMenu(
      id = "tabs",
      # [omitted code],
      tags$div(
        class = "sidebar-logo",
        boastUtils::psu_eberly_logo("reversed")
        )
      ),
    dashboardBody(
    tags$head(
      tags$link(rel = "stylesheet", type = "text/css",
        href = "https://educationshinyappteam.github.io/Style_Guide/theme/boast.css"),
      tabItems(
       # [omitted code]
      )  )  )  )  )
```

This will ensure that the Penn State logo gets properly used.

10.2 Colors

Your App needs to have a consistent color scheme throughout. The color scheme should be checked against colorblindness to meet WCAG 2.1 Level AA. You can do so at the Coloring for Colorblindness website. If you are following this Style Guide (as you should be) then the vast majority of this section will be automatically handled for you.

See Section 8.1.2 and Section 10.4.3 for additional information on the use of colors.

10.3 Text Styling

Text styling refers the non-content aspects of the text on the page, such as the use of italics, boldface, alignment, as well as font size and color.

You should let the centralized CSS file do the heavy lifting for text styling. (Again, using `boastApp` will help you.) However, for this to work properly, you will need to tag content appropriately. (See the section on HTML, Section 5.4.)

If you run into a situation where some element needs additional styling, **talk to Neil or Bob for help**. You might have come across an element that needs to get added the central CSS file or a bug.

10.3.1 Headings

Use the Heading Tags for the short fragments that define the structure of your App. If you find yourself enclosing a complete sentence in Heading tag, you ARE NOT using headings correctly. Notice how the headings in this Style

Guide aren't complete sentences; your App should mimic this. Full sentences appear as regular paragraph text (i.e., enclosed in `p()`) and not be a Heading.

10.3.2 Paragraph Text

If you enclose text that gives instructions or other information to your App's users in `p()` or `li()` (the later should be wrapped in either `tags$ol()` or `tags$ul()`), your App will understand how to style that text correctly. The central CSS file contains controls that set the base font size much larger than Shiny does natively as well as making text sizing dynamic. (This is important for making our apps mobile device friendly.) Again, using `boastApp` makes this process easier.

If you want to make a certain word or phrase italic, you will need to wrap that text in `tags$em()`. Similarly, if you want do the same with boldface, you'll use `tags$strong()`.

For example, this code:

```r
p(
  "When dealing with the ",
  tags$em("t"),
  "-distribution, we only have one parameter, the ",
  tags$strong("degrees of freedom"),
  "that we need to input."
)
```

Becomes:

> When dealing with the *t*-distribution, we only have one parameter, the **degrees of freedom** that we need to input.

Use italics (emphasis), and boldface (strong) sparingly.

10.3.3 Mathematics

For the most part, any mathematics you need displayed should be done using MathJax. Default to using inline typesetting with the `\\(` and `\\)` delimiters. If you need to use display style, you can use `\\[` and `\\]`. For the vast majority of mathematics, you'll wrap both inline and display style mathematics inside of a paragraph environment (`p()`).

If you're writing mathematics directly in your app, remember you'll need to escape the LaTeX commands by putting an extra backslash (\) in front; e.g., `\frac{3}{4}` would need to be `\\frac{3}{4}`.

If you're reading in mathematical text from an external CSV file, you do not need the extra backslash in the CSV file.

If you need assistance in figuring out how to type up mathematics, please talk to Neil, Matt, or Dennis.

Note: Double dollar sign delimiters are generally not recommended for displaying math as they can lead to unintended results. See: Writing Mathematics for MathJax.

10.3.4 [Game] Question Text

The text used as a question in a game should NOT be wrapped in a Heading tag; wrap the text in a paragraph tag.

10.3.5 Label Text (Buttons, Sliders, Other Inputs and Alerts)

By using the central CSS file, any text you included in/on buttons, dropdown menus, sliders, radio buttons, choices, and other inputs as well as alert messages and popups/rollovers, will automatically be styled correctly.

Do not use heading tags, the paragraph tag, italics/emphasis, or boldface/strong with input labels. Input labels should be written in sentence case (i.e., capitalize only the first word and any proper nouns).

You may use these tags with popups/rollovers.

10.3.6 Feedback and Hint Text

Again, let the central CSS file handle the styling of this type of text.

10.3.7 Text in `R` Plots

Unfortunately, any text in `R` plots does not get controlled by CSS. This means that you'll have to play around with the settings. Using the `ggplot2` package to make your plots (or other packages based upon the `grid` framework like `lattice`) will allow you to use the `theme` aspect to control text in your App.

Here is an example for how to do this:

```r
# Create a ggplot2 object
g1 <- ggplot2::ggplot(data=df, aes(x=x, y=y, color=grp))
# Add your layers, for example
g1 + ggplot2::geom_point()
# Use theme to control text size
g1 + ggplot2::theme(
  plot.caption = element_text(size = 18),
  text = element_text(size = 18)
  )
```

You will need to play around with the settings to find the appropriate value; text size 18 appears to work out well in many cases.

Note: The text in your plot might not behave well for dynamic re-sizing on different mobile devices.

10.3.8 Text Case

When writing text, you should use the appropriate case (i.e., which words you capitalize):

- Headings: use Title Case
- Paragraphs/Lists: use Sentence Case
- Input Labels: use Sentence Case
- Figure Captions: use Title Case
- Table Captions: use Title Case
- Graphs (see Section 10.4.1)
  - Titles: use Title Case
  - Axes: Use Sentence Case
  - Legends: Use Sentence Case
- Default: Sentence Case

10.4 Graphics

One of the most powerful tools we have in Statistics and Data Science is graphics. This includes images/pictures, graphs/plots, and tables. You will want to make sure that all graphical elements are appropriately sized in the Body. If there is text in a static image/picture, you'll need to make sure that the text is legible on a variety of screen sizes.

We've already discussed both issues of color and text size in plots. For additional considerations, please refer to the following readings (ordered from most important to least):

- Tufte-Fundamental Principles of Analytical Design
- Tufte-Chartjunk

- Kosslyn-Looking with the Eye and Mind

Remember, we always want to be modeling excellent graphing behaviors.

> All photographs can be fortified with words. –Dorothea Lange

> A picture is worth a thousand words…but which ones? –Unknown

Both of these quotations highlight that you need to include some text with your plots to help the user construct their understanding of what you're trying to show them.

10.4.1 Titles and Labels

Graph and Table titles should follow Title Case. Capitalize each word unless the word is "small" (e.g., of, an, etc.). The Title Case website can help you if you aren't sure. (This website also does other types of case such as camel case.)

10.4.2 Axes and Scales

R's default axes are terrible. They often do not fully cover the data and the have gaps between the axes. All this impedes the user's construction of meaning. Thus, you'll want to take control and stipulate the axes and scales to optimize what users get out of the plot. If you are providing multiple plots that the user is supposed to compare, make sure that they all use the same scaling and axes.

To force `ggplot2` to place (0, 0) in the lower-left corner and to control the scales, you will need to include the following:

```r
# Create the ggplot2 object
g1 <- ggplot2::ggplot(...)
# Add your layer
g1 + ggplot2::geom_point()
# Control axes and scale
## Multiplicative Scaling of the Horizontal (x) Axis
## Additive Scaling of the Vertical (y) Axis
g1 + ggplot2::scale_x_continuous(expand = expansion(mult = c(1,2), add = 0)) +
  scale_y_continuous(expand = expansion(mult = 0, add = c(0,0.05)))
```

10.4.3 Color and Plots in R

In R you can set color theme which you use in `ggplot2`. Here are two custom color palettes that you can use in your App. Additionally, the package `viridis` provides several additional color palettes which are improvements upon the default color scheme.

```r
# boastPalette is based on the Wong color blind set found at the above website.
boastPalette <- c("#0072B2","#D55E00","#009E73","#CE77A8",
    "#000000","#E69F00","#999999","#56B4E9","#CC79A7")

# psuPalette is based on Penn State's three official color palettes
# and checked at the above webite.
psuPalette <- c("#1E407C","#BC204B","#3EA39E","#E98300",
    "#999999","#AC8DCE","#F2665E","#99CC00")

# Both palettes get used in the order of what is listed.
```

To use these palettes (or ones from `viridis`) with a `ggplot2` object, you'll need to do the following

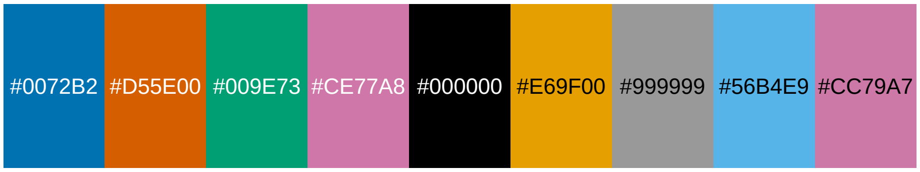


Figure 10.1: The Boast Palette

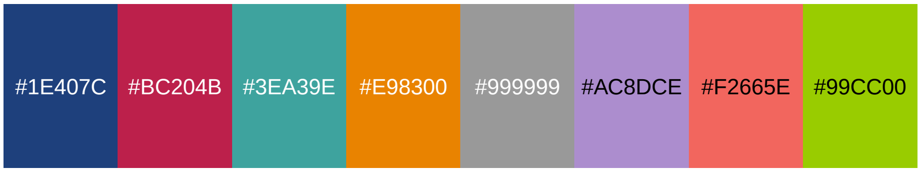


Figure 10.2: The PSU Palette

```r
# You will need to first add whichever palette line from above to your code
boastPalette <- c("#0072B2","#D55E00","#009E73","#CE77A8",
    "#000000","#E69F00","#999999","#56B4E9","#CC79A7")

# Create ggplot2 object
g1 <- ggplot2::ggplot(data = df,
                aes(x = x, y = y, color = grp, fill = grp))
# Add your layers
g1 + ggplot2::geom_points()
# Tell R to use your chosen palette
g1 + ggplot2::scale_color_manual(values=boastPalette)  # If you use "color" in aes
g1 + ggplot2::scale_fill_manual(values=boastPalette)  # If you use "fill" in aes
```

If you have more groups than eight/nine colors listed in the two palettes, consider reworking your examples as you could overwhelm the user with too many colors. (This also applies to using different shapes to plot points.)

With an eye towards accessibility, try not to use only color to denote a particular piece of information. Rather you might want to use color and shape.

10.4.4 Tables

Data tables can pose a challenge for individuals to comprehend. Just as a wall of text isn't conducive to helping a person understand what's going on, neither is a wall of data values. Thus, we need to be extreme judicious (picky) about incorporating data tables into any of our apps.

In web development there are two main types of tables: layout tables and data tables.

- Layout tables help control where different elements appear on the page.
- We need an additional distinction for data tables:
  - Summary Data Tables are tables that have summary information; typified by two-way tables (a.k.a. contingency tables or crosstabs) but might also include other things such as values of descriptive statistics stratified by groups.
  - Data Sets are an entire data object, presented in tabular format

Layout Tables should never be used in a BOAST App.

Data Sets should be displayed **as sparingly as possible**. In order to include a Data Set display, you will need to have identified an explicit learning goal/objective that necessitates the user digging through a data frame. If you can't identify such a learning goal, you should NOT include a data frame.

If the goal is to allow the user to look through the data set OR to have access to the data, then **give a link** to either the original source of the data (**preferred**) or for them to download the file.

Summary Data Tables can be used more often and can enrich the user's experience with your app. However, these must still be constructed in an appropriate manner.

Neither Data Sets nor Summary Data Tables should be inserted into your App as a picture. This is an big Accessibility violation. Use the directions below to create the appropriate type of data table.

10.4.4.1 Displaying Data Tables

Your first step is to create a data frame object in your `R` code.

- If you are displaying a data set (rare), then you will either need to read in the data or call that data frame. For this example, we'll be using the `mtcars` data frame that is part of `R`.
- If you are making a Summary Data Table, you'll need to either use `R` to calculate the values and store in a data frame or create a data frame yourself.

In either case, be sure you identify what columns you're going to use. If your original data file has 50 columns, but your App only makes use of 5, drop the other 45. Only display the columns that you actually use.

Your next step is to decide on where to put this display (e.g., inside an Exploration Tab or as a separate page). This will help you identify where in your App's UI section you need to put the appropriate code.

To ensure that your data table is accessible and responsive (i.e., mobile friendly), you will need to use the `DT` package.

```r
install.packages("DT")
# Be sure to include this in your library call
library(DT)
```

In your UI section, you'll need to use the following code, placed in the appropriate area:

```r
# [code omitted]
DT::DTOutput(outputId = "mtCars")
# [code omitted]
```

Then, in your Server section, you'll need to use the following code:

```r
# [code omitted]
# Prepare your data set with only the columns needed
carData <- mtcars[,c("mpg", "cyl", "hp", "gear", "wt")]

## Use Short but Meaningful Column Names
names(carData) <- c("MPG", "# of Cylinders", "Horsepower", "# of Gears", "Weight")

# Create the output data table
# Be sure to use the same name as you did in the UI
output$mtCars <- DT::renderDT(
  expr = carData,
  caption = "Motor Trend US Data, 1973-1974 Models", # Add a caption to your table
  style = "bootstrap4", # You must use this style
  rownames = TRUE,
  options = list( # You must use these options
    responsive = TRUE, # allows the data table to be mobile friendly
    scrollX = TRUE, # allows the user to scroll through a wide table
    columnDefs = list(  # These will set alignment of data values
      # Notice the use of ncol on your data frame; leave the 1 as is.
      list(className = 'dt-center', targets = 1:ncol(carData))
    )
  )
)
# [code omitted]
```

If you are making a Summary Data Table, you will need to follow the same process. If your data frame does not have row names, but instead a column with values acting as row names, you may replace the `rownames = TRUE` with `rownames = FALSE`; there should not be a column of sequential numbers on the left.

Column names **MUST** be simple and *meaningful* to the user. To this end, you should rename any columns that might have poor choices for names, just as we have done with the `mtcars` data. This includes using Greek characters in

isolation. You should not have any columns labeled μ or σ. Rather you need to use English words.

Note: getting mathematical expressions to render properly in graphical environments in `R` *is not as easy as in the paragraphs or headers of an app. Only certain graphing packages support limited mathematical expressions. The same is true for table generation packages.*

Again, try to use tables as infrequently as possible. Poorly constructed tables can create accessibility issues causing screen readers to poorly communicate tables to your users. If you run into problems and/or have questions, **talk to Neil and Bob**.

10.4.4.2 Additional Table Examples

We're including some additional Summary Data Table examples. For these examples, I'm going to make use of the `palmerpenguins` package of data sets.

```r
library(palmerpenguins)
library(psych)
library(DT)
library(tibble)

penStats <- psych::describeBy(
  x = penguins$body_mass_g,
  group = penguins$species,
  mat = TRUE, # Formats output appropriate for DT
  digits = 3 # sets the number of digits retained
)

# Picking which columns to keep
penStats <- penStats[, c("group1", "n", "mean", "sd", "median", "mad", "min", "max", "
                         "kurtosis")]
# Make the group1 column the row names
penStats <- tibble::remove_rownames(penStats)
penStats <- tibble::column_to_rownames(penStats,
                          var = "group1")
# Improve column names
names(penStats) <- c("Count", "SAM (g/penguin)", "SASD (g)", "Median (g)", "MAD (g)", "
                     "Max (g)", "Sample Skewness (g^3)", "Sample Excess Kurtosis (g^4)"
```

```r
# Make the Table
output$penguinSummary <- DT::renderDT(
  expr = penStats,
  caption = "Descriptive Stats for Palmer Penguins",
  style = "bootstrap4",
  rownames = TRUE,
  autoHideNavigation = TRUE,
  options = list(
    responsive = TRUE,
    scrollX = TRUE,
    paging = FALSE, # Set to False for small tables
     columnDefs = list(
       list(className = 'dt-center',
            targets = 1:ncol(penStats))
    )
  )
)
```

10.4.4.2.1 Summary Data Table of Descriptive Statistics

10.4.4.2.2 Summary Data Table for Output Table While this example is for an ANOVA table, you can build from this for other output tables. If you store the output of any call as an object, you can then use the structure function, `str` to investigate the output. Ultimately, you need something that is either a matrix or a data frame.

```
library(palmerpenguins)
library(psych)
library(DT)
library(tibble)
library(rstatix)
```

```
##
## Attaching package: 'rstatix'

## The following object is masked from 'package:stats':
##
##     filter
```

```
# This bad practice but I'm going to pretend that all assumptions are met
penModel <- aov(body_mass_g ~ species*sex, data = penguins)

anovaPen <- round(anova(penModel), 3)
# Rounding to truncate decimals
```

```
# Make the Table
output$penguinAnova <- DT::renderDT(
  expr = anovaPen,
  caption = "(Classical) ANOVA Table for Palmer Pengins",
  style = "bootstrap4",
  rownames = TRUE,
  options = list(
    responsive = TRUE,
    scrollX = TRUE,
    paging = FALSE, # Set to False for small tables
     columnDefs = list(
       list(className = 'dt-center',
            targets = 1:ncol(anovaPen))
    )
  )
)
```

10.4.5 Static Images

Static image refers to any image you're using in your App which is not produced by `R`. These are usually PNG or JPG/JPEG files which you end up calling in the UI portion of your code.

Within your App's folder/directory, there needs to be a sub-folder/directory called `www`. This is the place where you'll need to place ALL static image files.

10.4.5.1 Adding an Image

To include the image in your App, you'll need to make use of the image tag, `img`. When you run your App, Shiny automatically knows to check the `www` folder any time the `img` tag gets called. Here is an example supposing that the check mark image for correct answers is in the app's `www` folder:

```
#[code omitted]
div(align = "right",
    img(src = "check.PNG",
        alt = "Success, you are correct",
        width = 25, #these are in pixels
        height = 25,
        ))
#[code omitted]
```

You'll notice that we've wrapped the `img` call in a `div` call. The `div` call allows us to specify that we want the image to be right aligned; you could also do left or center. You can replace the `div` call with the paragraph environment and include text on either side, effectively making your image part of the text.

```
#[code omitted]
p("Check your answer here -->",
  img(src = "check.PNG",
      alt = "Success, you are correct",
      width = 25, height = 25),
  "<-- Check your answer here"),
#[code omitted]
```

10.4.5.2 Sizing and Positioning Your Image

All image files have a native size that is part of that file. For instance, the check mark image is 270 x 250 pixels. However, we overrode that that sizing with the `width` and `height` arguments. How did we decide on 25 x 25? Honestly, through guessing and checking. You'll need to think about how you're using the image and let that guide your decision making. There is no one size fits all solution. While finding an optimal size and position for your image can take some time, seeing bad settings is pretty obvious. Feel free to reach out to Neil and Bob for assistance.

10.4.6 Alt Text

Any graphical element you include in your App **MUST** have an alternative (assistive) text description ("alt text"). This provides a short description of what is in the image or plot for users who are visual impaired. (Tables, when properly formatted will handle this automatically.)

Here are several resources worth checking out:

- WebAIM Alternative Text Guide
- Penn State's Image ALT Text Page
- W3C's ALT Text Decision Tree

10.4.6.1 Adding Alt Text to Static Images

In the prior section on static images, you saw exactly how to set the alt text; here is a generic example:

```
#[code omitted]
img(src = "yourImage.PNG",
    alt = "Short description of what's in the pic",
    width = 25, height = 25)
#[code omitted]
```

10.4.6.2 Adding Alt Text Graphs

At this point in time (6/10/2020), adding alt text to plots generated in `R` is not as easy as for static pictures. While there are some potential changes coming down the pipe, there is not a firm time line for the addition of appropriate arguments to the existing `render*` functions. Therefore we will need to put in some (hopefully) temporary measures.

In particular, we will use Accessible Rich Internet Applications (ARIA) to assist us in writing some labels that will stand in place of formal alt text. To do this, you will need to make use of the following code:

```
# [code omitted]
# In the UI section, in the appropriate tabItem
plotOutput(outputId = plotID) # Look for lines like this
# Code for adding the aria label
tags$script(HTML(
  "$(document).ready(function() {
  document.getElementById('plotId').setAttribute('aria-label',
  `General description of the plot`)
  })"
))
# [code omitted]
```

Important things to note:

1. Place the `tags$script(HTML(...))` code right after each instance of `plotOutput`.
2. Copy the above code as formatted
3. Change the two (2) pieces for each particular plot
    a. Replace `plotId` (keep the single quotation marks in the code)
    b. Replace `General description of the plot` (keep the single quotation marks in the code)

Chapter 11

Design Style

Design Style is the second side of the Visual Appearance of every app. Here, we deal with visual aspects that go beyond the layout of your App. This includes issues of branding, color, how you make text look (font, size, emphasis), and graphics (plots, tables, and images).

11.1 PSU Branding

Given that we are all associated with Pennsylvania State University, we need to include the Penn State logo in each App. Rather than sticking the logo at the top of the Overview page, we are going to place the logo at the bottom of the sidebar. This has the benefit of having the logo appear throughout the entire App AND making the logo be as unobtrusive as possible.

In your UI section of `app.R` (or the `ui.R` file), at the end of the `dashboardSidebar()` section, you will need to include the following:

```
tags$div(class = "sidebar-logo",
         boastUtils::psu_eberly_logo("reversed"))
```

Here's how this code would look in context:

```
dashboardPage(
  skin = "blue",
  dashboardHeader(
    title = "title",
    # [omitted code]
  ),
  dashboardSidebar(
    width = 300,
    sidebarMenu(
      id = "tabs",
      # [omitted code],
      tags$div(
        class = "sidebar-logo",
        boastUtils::psu_eberly_logo("reversed")
        )
      ),
    dashboardBody(
    tags$head(
      tags$link(rel = "stylesheet", type = "text/css",
        href = "https://educationshinyappteam.github.io/Style_Guide/theme/boast.css"),
      tabItems(
       # [omitted code]
      )  )  )  )  )
```

This will ensure that the Penn State logo gets properly used.

11.2 Colors

Your App needs to have a consistent color scheme throughout. The color scheme should be checked against colorblindness to meet WCAG 2.1 Level AA. You can do so at the Coloring for Colorblindness website. If you are following this Style Guide (as you should be) then the vast majority of this section will be automatically handled for you.

See Section 8.1.2 and Section 10.4.3 for additional information on the use of colors.

11.3 Text Styling

Text styling refers the non-content aspects of the text on the page, such as the use of italics, boldface, alignment, as well as font size and color.

You should let the centralized CSS file do the heavy lifting for text styling. (Again, using `boastApp` will help you.) However, for this to work properly, you will need to tag content appropriately. (See the section on HTML, Section 5.4.)

If you run into a situation where some element needs additional styling, **talk to Neil or Bob for help**. You might have come across an element that needs to get added the central CSS file or a bug.

11.3.1 Headings

Use the Heading Tags for the short fragments that define the structure of your App. If you find yourself enclosing a complete sentence in Heading tag, you ARE NOT using headings correctly. Notice how the headings in this Style

Guide aren't complete sentences; your App should mimic this. Full sentences appear as regular paragraph text (i.e., enclosed in `p()`) and not be a Heading.

11.3.2 Paragraph Text

If you enclose text that gives instructions or other information to your App's users in `p()` or `li()` (the later should be wrapped in either `tags$ol()` or `tags$ul()`), your App will understand how to style that text correctly. The central CSS file contains controls that set the base font size much larger than Shiny does natively as well as making text sizing dynamic. (This is important for making our apps mobile device friendly.) Again, using `boastApp` makes this process easier.

If you want to make a certain word or phrase italic, you will need to wrap that text in `tags$em()`. Similarly, if you want do the same with boldface, you'll use `tags$strong()`.

For example, this code:

```
p(
  "When dealing with the ",
  tags$em("t"),
  "-distribution, we only have one parameter, the ",
  tags$strong("degrees of freedom"),
  "that we need to input."
)
```

Becomes:

> When dealing with the *t*-distribution, we only have one parameter, the **degrees of freedom** that we need to input.

Use italics (emphasis), and boldface (strong) sparingly.

11.3.3 Mathematics

For the most part, any mathematics you need displayed should be done using MathJax. Default to using inline typesetting with the `\\(` and `\\)` delimiters. If you need to use display style, you can use `\\[` and `\\]`. For the vast majority of mathematics, you'll wrap both inline and display style mathematics inside of a paragraph environment (`p()`).

If you're writing mathematics directly in your app, remember you'll need to escape the LaTeX commands by putting an extra backslash (\) in front; e.g., `\frac{3}{4}` would need to be `\\frac{3}{4}`.

If you're reading in mathematical text from an external CSV file, you do not need the extra backslash in the CSV file.

If you need assistance in figuring out how to type up mathematics, please talk to Neil, Matt, or Dennis.

Note: Double dollar sign delimiters are generally not recommended for displaying math as they can lead to unintended results. See: Writing Mathematics for MathJax.

11.3.4 [Game] Question Text

The text used as a question in a game should NOT be wrapped in a Heading tag; wrap the text in a paragraph tag.

11.3.5 Label Text (Buttons, Sliders, Other Inputs and Alerts)

By using the central CSS file, any text you included in/on buttons, dropdown menus, sliders, radio buttons, choices, and other inputs as well as alert messages and popups/rollovers, will automatically be styled correctly.

Do not use heading tags, the paragraph tag, italics/emphasis, or boldface/strong with input labels. Input labels should be written in sentence case (i.e., capitalize only the first word and any proper nouns).

You may use these tags with popups/rollovers.

11.3.6 Feedback and Hint Text

Again, let the central CSS file handle the styling of this type of text.

11.3.7 Text in `R` Plots

Unfortunately, any text in `R` plots does not get controlled by CSS. This means that you'll have to play around with the settings. Using the `ggplot2` package to make your plots (or other packages based upon the `grid` framework like `lattice`) will allow you to use the `theme` aspect to control text in your App.

Here is an example for how to do this:

```r
# Create a ggplot2 object
g1 <- ggplot2::ggplot(data=df, aes(x=x, y=y, color=grp))
# Add your layers, for example
g1 + ggplot2::geom_point()
# Use theme to control text size
g1 + ggplot2::theme(
  plot.caption = element_text(size = 18),
  text = element_text(size = 18)
  )
```

You will need to play around with the settings to find the appropriate value; text size 18 appears to work out well in many cases.

Note: The text in your plot might not behave well for dynamic re-sizing on different mobile devices.

11.3.8 Text Case

When writing text, you should use the appropriate case (i.e., which words you capitalize):

- Headings: use Title Case
- Paragraphs/Lists: use Sentence Case
- Input Labels: use Sentence Case
- Figure Captions: use Title Case
- Table Captions: use Title Case
- Graphs (see Section 10.4.1)
  - Titles: use Title Case
  - Axes: Use Sentence Case
  - Legends: Use Sentence Case
- Default: Sentence Case

11.4 Graphics

One of the most powerful tools we have in Statistics and Data Science is graphics. This includes images/pictures, graphs/plots, and tables. You will want to make sure that all graphical elements are appropriately sized in the Body. If there is text in a static image/picture, you'll need to make sure that the text is legible on a variety of screen sizes.

We've already discussed both issues of color and text size in plots. For additional considerations, please refer to the following readings (ordered from most important to least):

- Tufte-Fundamental Principles of Analytical Design
- Tufte-Chartjunk

- Kosslyn-Looking with the Eye and Mind

Remember, we always want to be modeling excellent graphing behaviors.

> All photographs can be fortified with words. –Dorothea Lange

> A picture is worth a thousand words…but which ones? –Unknown

Both of these quotations highlight that you need to include some text with your plots to help the user construct their understanding of what you're trying to show them.

11.4.1 Titles and Labels

Graph and Table titles should follow Title Case. Capitalize each word unless the word is "small" (e.g., of, an, etc.). The Title Case website can help you if you aren't sure. (This website also does other types of case such as camel case.)

11.4.2 Axes and Scales

R's default axes are terrible. They often do not fully cover the data and the have gaps between the axes. All this impedes the user's construction of meaning. Thus, you'll want to take control and stipulate the axes and scales to optimize what users get out of the plot. If you are providing multiple plots that the user is supposed to compare, make sure that they all use the same scaling and axes.

To force ggplot2 to place (0, 0) in the lower-left corner and to control the scales, you will need to include the following:

```r
# Create the ggplot2 object
g1 <- ggplot2::ggplot(...)
# Add your layer
g1 + ggplot2::geom_point()
# Control axes and scale
## Multiplicative Scaling of the Horizontal (x) Axis
## Additive Scaling of the Vertical (y) Axis
g1 + ggplot2::scale_x_continuous(expand = expansion(mult = c(1,2), add = 0)) +
  scale_y_continuous(expand = expansion(mult = 0, add = c(0,0.05)))
```

11.4.3 Color and Plots in R

In R you can set color theme which you use in ggplot2. Here are two custom color palettes that you can use in your App. Additionally, the package viridis provides several additional color palettes which are improvements upon the default color scheme.

```r
# boastPalette is based on the Wong color blind set found at the above website.
boastPalette <- c("#0072B2","#D55E00","#009E73","#CE77A8",
    "#000000","#E69F00","#999999","#56B4E9","#CC79A7")

# psuPalette is based on Penn State's three official color palettes
# and checked at the above webite.
psuPalette <- c("#1E407C","#BC204B","#3EA39E","#E98300",
    "#999999","#AC8DCE","#F2665E","#99CC00")

# Both palettes get used in the order of what is listed.
```

To use these palettes (or ones from viridis) with a ggplot2 object, you'll need to do the following

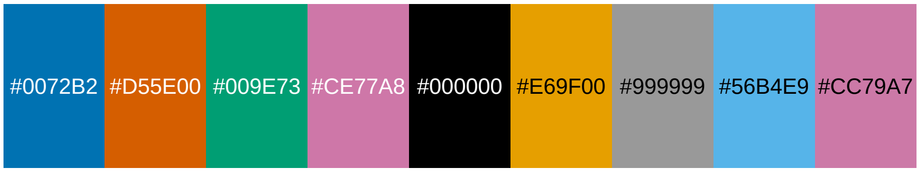


Figure 11.1: The Boast Palette

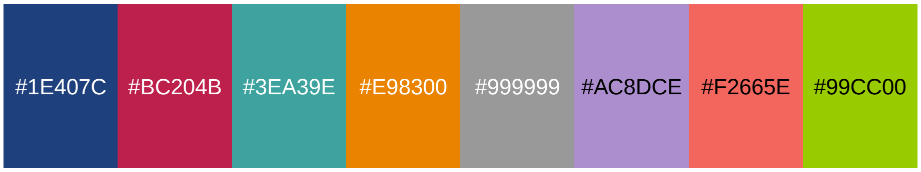


Figure 11.2: The PSU Palette

```r
# You will need to first add whichever palette line from above to your code
boastPalette <- c("#0072B2","#D55E00","#009E73","#CE77A8",
    "#000000","#E69F00","#999999","#56B4E9","#CC79A7")

# Create ggplot2 object
g1 <- ggplot2::ggplot(data = df,
                aes(x = x, y = y, color = grp, fill = grp))
# Add your layers
g1 + ggplot2::geom_points()
# Tell R to use your chosen palette
g1 + ggplot2::scale_color_manual(values=boastPalette)  # If you use "color" in aes
g1 + ggplot2::scale_fill_manual(values=boastPalette)  # If you use "fill" in aes
```

If you have more groups than eight/nine colors listed in the two palettes, consider reworking your examples as you could overwhelm the user with too many colors. (This also applies to using different shapes to plot points.)

With an eye towards accessibility, try not to use only color to denote a particular piece of information. Rather you might want to use color and shape.

11.4.4 Tables

Data tables can pose a challenge for individuals to comprehend. Just as a wall of text isn't conducive to helping a person understand what's going on, neither is a wall of data values. Thus, we need to be extreme judicious (picky) about incorporating data tables into any of our apps.

In web development there are two main types of tables: layout tables and data tables.

- Layout tables help control where different elements appear on the page.
- We need an additional distinction for data tables:
    - Summary Data Tables are tables that have summary information; typified by two-way tables (a.k.a. contingency tables or crosstabs) but might also include other things such as values of descriptive statistics stratified by groups.
    - Data Sets are an entire data object, presented in tabular format

Layout Tables should never be used in a BOAST App.

Data Sets should be displayed **as sparingly as possible**. In order to include a Data Set display, you will need to have identified an explicit learning goal/objective that necessitates the user digging through a data frame. If you can't identify such a learning goal, you should NOT include a data frame.

If the goal is to allow the user to look through the data set OR to have access to the data, then **give a link** to either the original source of the data (**preferred**) or for them to download the file.

Summary Data Tables can be used more often and can enrich the user's experience with your app. However, these must still be constructed in an appropriate manner.

Neither Data Sets nor Summary Data Tables should be inserted into your App as a picture. This is an big Accessibility violation. Use the directions below to create the appropriate type of data table.

11.4.4.1 Displaying Data Tables

Your first step is to create a data frame object in your `R` code.

- If you are displaying a data set (rare), then you will either need to read in the data or call that data frame. For this example, we'll be using the `mtcars` data frame that is part of `R`.
- If you are making a Summary Data Table, you'll need to either use `R` to calculate the values and store in a data frame or create a data frame yourself.

In either case, be sure you identify what columns you're going to use. If your original data file has 50 columns, but your App only makes use of 5, drop the other 45. Only display the columns that you actually use.

Your next step is to decide on where to put this display (e.g., inside an Exploration Tab or as a separate page). This will help you identify where in your App's UI section you need to put the appropriate code.

To ensure that your data table is accessible and responsive (i.e., mobile friendly), you will need to use the `DT` package.

```
install.packages("DT")
# Be sure to include this in your library call
library(DT)
```

In your UI section, you'll need to use the following code, placed in the appropriate area:

```
# [code omitted]
DT::DTOutput(outputId = "mtCars")
# [code omitted]
```

Then, in your Server section, you'll need to use the following code:

```
# [code omitted]
# Prepare your data set with only the columns needed
carData <- mtcars[,c("mpg", "cyl", "hp", "gear", "wt")]

## Use Short but Meaningful Column Names
names(carData) <- c("MPG", "# of Cylinders", "Horsepower", "# of Gears", "Weight")

# Create the output data table
# Be sure to use the same name as you did in the UI
output$mtCars <- DT::renderDT(
  expr = carData,
  caption = "Motor Trend US Data, 1973-1974 Models", # Add a caption to your table
  style = "bootstrap4", # You must use this style
  rownames = TRUE,
  options = list( # You must use these options
    responsive = TRUE, # allows the data table to be mobile friendly
    scrollX = TRUE, # allows the user to scroll through a wide table
    columnDefs = list(  # These will set alignment of data values
      # Notice the use of ncol on your data frame; leave the 1 as is.
      list(className = 'dt-center', targets = 1:ncol(carData))
    )
  )
)
# [code omitted]
```

If you are making a Summary Data Table, you will need to follow the same process. If your data frame does not have row names, but instead a column with values acting as row names, you may replace the `rownames = TRUE` with `rownames = FALSE`; there should not be a column of sequential numbers on the left.

Column names **MUST** be simple and *meaningful* to the user. To this end, you should rename any columns that might have poor choices for names, just as we have done with the `mtcars` data. This includes using Greek characters in

isolation. You should not have any columns labeled μ or σ. Rather you need to use English words.

Note: getting mathematical expressions to render properly in graphical environments in `R` *is not as easy as in the paragraphs or headers of an app. Only certain graphing packages support limited mathematical expressions. The same is true for table generation packages.*

Again, try to use tables as infrequently as possible. Poorly constructed tables can create accessibility issues causing screen readers to poorly communicate tables to your users. If you run into problems and/or have questions, **talk to Neil and Bob**.

11.4.4.2 Additional Table Examples

We're including some additional Summary Data Table examples. For these examples, I'm going to make use of the `palmerpenguins` package of data sets.

```r
library(palmerpenguins)
library(psych)
library(DT)
library(tibble)

penStats <- psych::describeBy(
  x = penguins$body_mass_g,
  group = penguins$species,
  mat = TRUE, # Formats output appropriate for DT
  digits = 3 # sets the number of digits retained
)

# Picking which columns to keep
penStats <- penStats[, c("group1", "n", "mean", "sd", "median", "mad", "min", "max", "
                         "kurtosis")]
# Make the group1 column the row names
penStats <- tibble::remove_rownames(penStats)
penStats <- tibble::column_to_rownames(penStats,
                          var = "group1")
# Improve column names
names(penStats) <- c("Count", "SAM (g/penguin)", "SASD (g)", "Median (g)", "MAD (g)", "
                     "Max (g)", "Sample Skewness (g^3)", "Sample Excess Kurtosis (g^4)"
```

```r
# Make the Table
output$penguinSummary <- DT::renderDT(
  expr = penStats,
  caption = "Descriptive Stats for Palmer Penguins",
  style = "bootstrap4",
  rownames = TRUE,
  autoHideNavigation = TRUE,
  options = list(
    responsive = TRUE,
    scrollX = TRUE,
    paging = FALSE, # Set to False for small tables
     columnDefs = list(
       list(className = 'dt-center',
            targets = 1:ncol(penStats))
    )
  )
)
```

11.4.4.2.1 Summary Data Table of Descriptive Statistics

11.4.4.2.2 Summary Data Table for Output Table While this example is for an ANOVA table, you can build from this for other output tables. If you store the output of any call as an object, you can then use the structure function, `str` to investigate the output. Ultimately, you need something that is either a matrix or a data frame.

```
library(palmerpenguins)
library(psych)
library(DT)
library(tibble)
library(rstatix)

# This bad practice but I'm going to pretend that all assumptions are met
penModel <- aov(body_mass_g ~ species*sex, data = penguins)

anovaPen <- round(anova(penModel), 3)
# Rounding to truncate decimals
```

```
# Make the Table
output$penguinAnova <- DT::renderDT(
  expr = anovaPen,
  caption = "(Classical) ANOVA Table for Palmer Pengins",
  style = "bootstrap4",
  rownames = TRUE,
  options = list(
    responsive = TRUE,
    scrollX = TRUE,
    paging = FALSE, # Set to False for small tables
     columnDefs = list(
       list(className = 'dt-center',
            targets = 1:ncol(anovaPen))
    )
  )
)
```

11.4.5 Static Images

Static image refers to any image you're using in your App which is not produced by `R`. These are usually PNG or JPG/JPEG files which you end up calling in the UI portion of your code.

Within your App's folder/directory, there needs to be a sub-folder/directory called `www`. This is the place where you'll need to place ALL static image files.

11.4.5.1 Adding an Image

To include the image in your App, you'll need to make use of the image tag, `img`. When you run your App, Shiny automatically knows to check the `www` folder any time the `img` tag gets called. Here is an example supposing that the check mark image for correct answers is in the app's `www` folder:

```
#[code omitted]
div(align = "right",
    img(src = "check.PNG",
        alt = "Success, you are correct",
        width = 25, #these are in pixels
        height = 25,
        ))
#[code omitted]
```

You'll notice that we've wrapped the `img` call in a `div` call. The `div` call allows us to specify that we want the image to be right aligned; you could also do left or center. You can replace the `div` call with the paragraph environment and include text on either side, effectively making your image part of the text.

```
#[code omitted]
p("Check your answer here -->",
  img(src = "check.PNG",
      alt = "Success, you are correct",
      width = 25, height = 25),
  "<-- Check your answer here"),
#[code omitted]
```

11.4.5.2 Sizing and Positioning Your Image

All image files have a native size that is part of that file. For instance, the check mark image is 270 x 250 pixels. However, we overrode that that sizing with the `width` and `height` arguments. How did we decide on 25 x 25? Honestly, through guessing and checking. You'll need to think about how you're using the image and let that guide your decision making. There is no one size fits all solution. While finding an optimal size and position for your image can take some time, seeing bad settings is pretty obvious. Feel free to reach out to Neil and Bob for assistance.

11.4.6 Alt Text

Any graphical element you include in your App **MUST** have an alternative (assistive) text description ("alt text"). This provides a short description of what is in the image or plot for users who are visual impaired. (Tables, when properly formatted will handle this automatically.)

Here are several resources worth checking out:

- WebAIM Alternative Text Guide
- Penn State's Image ALT Text Page
- W3C's ALT Text Decision Tree

11.4.6.1 Adding Alt Text to Static Images

In the prior section on static images, you saw exactly how to set the alt text; here is a generic example:

```
#[code omitted]
img(src = "yourImage.PNG",
    alt = "Short description of what's in the pic",
    width = 25, height = 25)
#[code omitted]
```

11.4.6.2 Adding Alt Text Graphs

At this point in time (6/10/2020), adding alt text to plots generated in `R` is not as easy as for static pictures. While there are some potential changes coming down the pipe, there is not a firm time line for the addition of appropriate arguments to the existing `render*` functions. Therefore we will need to put in some (hopefully) temporary measures.

In particular, we will use Accessible Rich Internet Applications (ARIA) to assist us in writing some labels that will stand in place of formal alt text. To do this, you will need to make use of the following code:

```
# [code omitted]
# In the UI section, in the appropriate tabItem
plotOutput(outputId = plotID) # Look for lines like this
# Code for adding the aria label
tags$script(HTML(
  "$(document).ready(function() {
```

```
  document.getElementById('plotId').setAttribute('aria-label',
  `General description of the plot`)
  })"
))
# [code omitted]
```

Important things to note:

1. Place the `tags$script(HTML(...))` code right after each instance of `plotOutput`.
2. Copy the above code as formatted
3. Change the two (2) pieces for each particular plot
    a. Replace `plotId` (keep the single quotation marks in the code)
    b. Replace `General description of the plot` (keep the single quotation marks in the code)

Chapter 12

Design Style

Design Style is the second side of the Visual Appearance of every app. Here, we deal with visual aspects that go beyond the layout of your App. This includes issues of branding, color, how you make text look (font, size, emphasis), and graphics (plots, tables, and images).

12.1 PSU Branding

Given that we are all associated with Pennsylvania State University, we need to include the Penn State logo in each App. Rather than sticking the logo at the top of the Overview page, we are going to place the logo at the bottom of the sidebar. This has the benefit of having the logo appear throughout the entire App AND making the logo be as unobtrusive as possible.

In your UI section of `app.R` (or the `ui.R` file), at the end of the `dashboardSidebar()` section, you will need to include the following:

```
tags$div(class = "sidebar-logo",
         boastUtils::psu_eberly_logo("reversed"))
```

Here's how this code would look in context:

```
dashboardPage(
  skin = "blue",
  dashboardHeader(
    title = "title",
    # [omitted code]
  ),
  dashboardSidebar(
    width = 300,
    sidebarMenu(
     id = "tabs",
     # [omitted code],
     tags$div(
      class = "sidebar-logo",
      boastUtils::psu_eberly_logo("reversed")
      )
     ),
    dashboardBody(
    tags$head(
     tags$link(rel = "stylesheet", type = "text/css",
      href = "https://educationshinyappteam.github.io/Style_Guide/theme/boast.css"),
     tabItems(
      # [omitted code]
     ) ) ) ) )
```

This will ensure that the Penn State logo gets properly used.

12.2 Colors

Your App needs to have a consistent color scheme throughout. The color scheme should be checked against colorblindness to meet WCAG 2.1 Level AA. You can do so at the Coloring for Colorblindness website. If you are following this Style Guide (as you should be) then the vast majority of this section will be automatically handled for you.

See Section 8.1.2 and Section 10.4.3 for additional information on the use of colors.

12.3 Text Styling

Text styling refers the non-content aspects of the text on the page, such as the use of italics, boldface, alignment, as well as font size and color.

You should let the centralized CSS file do the heavy lifting for text styling. (Again, using `boastApp` will help you.) However, for this to work properly, you will need to tag content appropriately. (See the section on HTML, Section 5.4.)

If you run into a situation where some element needs additional styling, **talk to Neil or Bob for help**. You might have come across an element that needs to get added the central CSS file or a bug.

12.3.1 Headings

Use the Heading Tags for the short fragments that define the structure of your App. If you find yourself enclosing a complete sentence in Heading tag, you ARE NOT using headings correctly. Notice how the headings in this Style

Guide aren't complete sentences; your App should mimic this. Full sentences appear as regular paragraph text (i.e., enclosed in `p()`) and not be a Heading.

12.3.2 Paragraph Text

If you enclose text that gives instructions or other information to your App's users in `p()` or `li()` (the later should be wrapped in either `tags$ol()` or `tags$ul()`), your App will understand how to style that text correctly. The central CSS file contains controls that set the base font size much larger than Shiny does natively as well as making text sizing dynamic. (This is important for making our apps mobile device friendly.) Again, using `boastApp` makes this process easier.

If you want to make a certain word or phrase italic, you will need to wrap that text in `tags$em()`. Similarly, if you want do the same with boldface, you'll use `tags$strong()`.

For example, this code:

```r
p(
  "When dealing with the ",
  tags$em("t"),
  "-distribution, we only have one parameter, the ",
  tags$strong("degrees of freedom"),
  "that we need to input."
)
```

Becomes:

> When dealing with the *t*-distribution, we only have one parameter, the **degrees of freedom** that we need to input.

Use italics (emphasis), and boldface (strong) sparingly.

12.3.3 Mathematics

For the most part, any mathematics you need displayed should be done using MathJax. Default to using inline typesetting with the `\\(` and `\\)` delimiters. If you need to use display style, you can use `\\[` and `\\]`. For the vast majority of mathematics, you'll wrap both inline and display style mathematics inside of a paragraph environment (`p()`).

If you're writing mathematics directly in your app, remember you'll need to escape the LaTeX commands by putting an extra backslash (\) in front; e.g., `\frac{3}{4}` would need to be `\\frac{3}{4}`.

If you're reading in mathematical text from an external CSV file, you do not need the extra backslash in the CSV file.

If you need assistance in figuring out how to type up mathematics, please talk to Neil, Matt, or Dennis.

Note: Double dollar sign delimiters are generally not recommended for displaying math as they can lead to unintended results. See: Writing Mathematics for MathJax.

12.3.4 [Game] Question Text

The text used as a question in a game should NOT be wrapped in a Heading tag; wrap the text in a paragraph tag.

12.3.5 Label Text (Buttons, Sliders, Other Inputs and Alerts)

By using the central CSS file, any text you included in/on buttons, dropdown menus, sliders, radio buttons, choices, and other inputs as well as alert messages and popups/rollovers, will automatically be styled correctly.

Do not use heading tags, the paragraph tag, italics/emphasis, or boldface/strong with input labels. Input labels should be written in sentence case (i.e., capitalize only the first word and any proper nouns).

You may use these tags with popups/rollovers.

12.3.6 Feedback and Hint Text

Again, let the central CSS file handle the styling of this type of text.

12.3.7 Text in `R` Plots

Unfortunately, any text in `R` plots does not get controlled by CSS. This means that you'll have to play around with the settings. Using the `ggplot2` package to make your plots (or other packages based upon the `grid` framework like `lattice`) will allow you to use the `theme` aspect to control text in your App.

Here is an example for how to do this:

```r
# Create a ggplot2 object
g1 <- ggplot2::ggplot(data=df, aes(x=x, y=y, color=grp))
# Add your layers, for example
g1 + ggplot2::geom_point()
# Use theme to control text size
g1 + ggplot2::theme(
  plot.caption = element_text(size = 18),
  text = element_text(size = 18)
  )
```

You will need to play around with the settings to find the appropriate value; text size 18 appears to work out well in many cases.

Note: The text in your plot might not behave well for dynamic re-sizing on different mobile devices.

12.3.8 Text Case

When writing text, you should use the appropriate case (i.e., which words you capitalize):

- Headings: use Title Case
- Paragraphs/Lists: use Sentence Case
- Input Labels: use Sentence Case
- Figure Captions: use Title Case
- Table Captions: use Title Case
- Graphs (see Section 10.4.1)
  - Titles: use Title Case
  - Axes: Use Sentence Case
  - Legends: Use Sentence Case
- Default: Sentence Case

12.4 Graphics

One of the most powerful tools we have in Statistics and Data Science is graphics. This includes images/pictures, graphs/plots, and tables. You will want to make sure that all graphical elements are appropriately sized in the Body. If there is text in a static image/picture, you'll need to make sure that the text is legible on a variety of screen sizes.

We've already discussed both issues of color and text size in plots. For additional considerations, please refer to the following readings (ordered from most important to least):

- Tufte-Fundamental Principles of Analytical Design
- Tufte-Chartjunk

- Kosslyn-Looking with the Eye and Mind

Remember, we always want to be modeling excellent graphing behaviors.

> All photographs can be fortified with words. –Dorothea Lange

> A picture is worth a thousand words…but which ones? –Unknown

Both of these quotations highlight that you need to include some text with your plots to help the user construct their understanding of what you're trying to show them.

12.4.1 Titles and Labels

Graph and Table titles should follow Title Case. Capitalize each word unless the word is "small" (e.g., of, an, etc.). The Title Case website can help you if you aren't sure. (This website also does other types of case such as camel case.)

12.4.2 Axes and Scales

R's default axes are terrible. They often do not fully cover the data and the have gaps between the axes. All this impedes the user's construction of meaning. Thus, you'll want to take control and stipulate the axes and scales to optimize what users get out of the plot. If you are providing multiple plots that the user is supposed to compare, make sure that they all use the same scaling and axes.

To force ggplot2 to place (0, 0) in the lower-left corner and to control the scales, you will need to include the following:

```r
# Create the ggplot2 object
g1 <- ggplot2::ggplot(...)
# Add your layer
g1 + ggplot2::geom_point()
# Control axes and scale
## Multiplicative Scaling of the Horizontal (x) Axis
## Additive Scaling of the Vertical (y) Axis
g1 + ggplot2::scale_x_continuous(expand = expansion(mult = c(1,2), add = 0)) +
  scale_y_continuous(expand = expansion(mult = 0, add = c(0,0.05)))
```

12.4.3 Color and Plots in R

In R you can set color theme which you use in ggplot2. Here are two custom color palettes that you can use in your App. Additionally, the package viridis provides several additional color palettes which are improvements upon the default color scheme.

```r
# boastPalette is based on the Wong color blind set found at the above website.
boastPalette <- c("#0072B2","#D55E00","#009E73","#CE77A8",
    "#000000","#E69F00","#999999","#56B4E9","#CC79A7")

# psuPalette is based on Penn State's three official color palettes
# and checked at the above webite.
psuPalette <- c("#1E407C","#BC204B","#3EA39E","#E98300",
    "#999999","#AC8DCE","#F2665E","#99CC00")

# Both palettes get used in the order of what is listed.
```

To use these palettes (or ones from viridis) with a ggplot2 object, you'll need to do the following

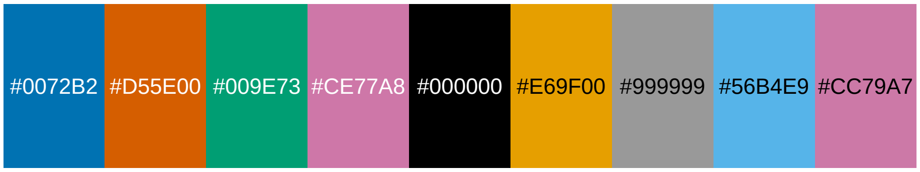


Figure 12.1: The Boast Palette

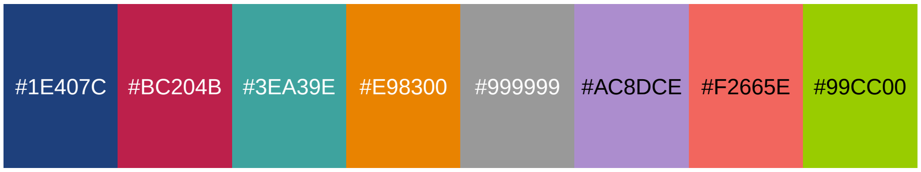


Figure 12.2: The PSU Palette

```r
# You will need to first add whichever palette line from above to your code
boastPalette <- c("#0072B2","#D55E00","#009E73","#CE77A8",
    "#000000","#E69F00","#999999","#56B4E9","#CC79A7")

# Create ggplot2 object
g1 <- ggplot2::ggplot(data = df,
                aes(x = x, y = y, color = grp, fill = grp))
# Add your layers
g1 + ggplot2::geom_points()
# Tell R to use your chosen palette
g1 + ggplot2::scale_color_manual(values=boastPalette)  # If you use "color" in aes
g1 + ggplot2::scale_fill_manual(values=boastPalette)  # If you use "fill" in aes
```

If you have more groups than eight/nine colors listed in the two palettes, consider reworking your examples as you could overwhelm the user with too many colors. (This also applies to using different shapes to plot points.)

With an eye towards accessibility, try not to use only color to denote a particular piece of information. Rather you might want to use color and shape.

12.4.4 Tables

Data tables can pose a challenge for individuals to comprehend. Just as a wall of text isn't conducive to helping a person understand what's going on, neither is a wall of data values. Thus, we need to be extreme judicious (picky) about incorporating data tables into any of our apps.

In web development there are two main types of tables: layout tables and data tables.

- Layout tables help control where different elements appear on the page.
- We need an additional distinction for data tables:
  - Summary Data Tables are tables that have summary information; typified by two-way tables (a.k.a. contingency tables or crosstabs) but might also include other things such as values of descriptive statistics stratified by groups.
  - Data Sets are an entire data object, presented in tabular format

Layout Tables should never be used in a BOAST App.

Data Sets should be displayed **as sparingly as possible**. In order to include a Data Set display, you will need to have identified an explicit learning goal/objective that necessitates the user digging through a data frame. If you can't identify such a learning goal, you should NOT include a data frame.

If the goal is to allow the user to look through the data set OR to have access to the data, then **give a link** to either the original source of the data (**preferred**) or for them to download the file.

Summary Data Tables can be used more often and can enrich the user's experience with your app. However, these must still be constructed in an appropriate manner.

Neither Data Sets nor Summary Data Tables should be inserted into your App as a picture. This is an big Accessibility violation. Use the directions below to create the appropriate type of data table.

12.4.4.1 Displaying Data Tables

Your first step is to create a data frame object in your `R` code.

- If you are displaying a data set (rare), then you will either need to read in the data or call that data frame. For this example, we'll be using the `mtcars` data frame that is part of `R`.
- If you are making a Summary Data Table, you'll need to either use `R` to calculate the values and store in a data frame or create a data frame yourself.

In either case, be sure you identify what columns you're going to use. If your original data file has 50 columns, but your App only makes use of 5, drop the other 45. Only display the columns that you actually use.

Your next step is to decide on where to put this display (e.g., inside an Exploration Tab or as a separate page). This will help you identify where in your App's UI section you need to put the appropriate code.

To ensure that your data table is accessible and responsive (i.e., mobile friendly), you will need to use the `DT` package.

```r
install.packages("DT")
# Be sure to include this in your library call
library(DT)
```

In your UI section, you'll need to use the following code, placed in the appropriate area:

```r
# [code omitted]
DT::DTOutput(outputId = "mtCars")
# [code omitted]
```

Then, in your Server section, you'll need to use the following code:

```r
# [code omitted]
# Prepare your data set with only the columns needed
carData <- mtcars[,c("mpg", "cyl", "hp", "gear", "wt")]

## Use Short but Meaningful Column Names
names(carData) <- c("MPG", "# of Cylinders", "Horsepower", "# of Gears", "Weight")

# Create the output data table
# Be sure to use the same name as you did in the UI
output$mtCars <- DT::renderDT(
  expr = carData,
  caption = "Motor Trend US Data, 1973-1974 Models", # Add a caption to your table
  style = "bootstrap4", # You must use this style
  rownames = TRUE,
  options = list( # You must use these options
    responsive = TRUE, # allows the data table to be mobile friendly
    scrollX = TRUE, # allows the user to scroll through a wide table
    columnDefs = list(  # These will set alignment of data values
      # Notice the use of ncol on your data frame; leave the 1 as is.
      list(className = 'dt-center', targets = 1:ncol(carData))
    )
  )
)
# [code omitted]
```

If you are making a Summary Data Table, you will need to follow the same process. If your data frame does not have row names, but instead a column with values acting as row names, you may replace the `rownames = TRUE` with `rownames = FALSE`; there should not be a column of sequential numbers on the left.

Column names **MUST** be simple and *meaningful* to the user. To this end, you should rename any columns that might have poor choices for names, just as we have done with the `mtcars` data. This includes using Greek characters in

isolation. You should not have any columns labeled μ or σ. Rather you need to use English words.

Note: getting mathematical expressions to render properly in graphical environments in `R` is not as easy as in the paragraphs or headers of an app. Only certain graphing packages support limited mathematical expressions. The same is true for table generation packages.

Again, try to use tables as infrequently as possible. Poorly constructed tables can create accessibility issues causing screen readers to poorly communicate tables to your users. If you run into problems and/or have questions, **talk to Neil and Bob**.

12.4.4.2 Additional Table Examples

We're including some additional Summary Data Table examples. For these examples, I'm going to make use of the `palmerpenguins` package of data sets.

```
library(palmerpenguins)
library(psych)
library(DT)
library(tibble)

penStats <- psych::describeBy(
  x = penguins$body_mass_g,
  group = penguins$species,
  mat = TRUE, # Formats output appropriate for DT
  digits = 3 # sets the number of digits retained
)

# Picking which columns to keep
penStats <- penStats[, c("group1", "n", "mean", "sd", "median", "mad", "min", "max", "s
                         "kurtosis")]
# Make the group1 column the row names
penStats <- tibble::remove_rownames(penStats)
penStats <- tibble::column_to_rownames(penStats,
                          var = "group1")
# Improve column names
names(penStats) <- c("Count", "SAM (g/penguin)", "SASD (g)", "Median (g)", "MAD (g)", "
                     "Max (g)", "Sample Skewness (g^3)", "Sample Excess Kurtosis (g^4)"
```

```
# Make the Table
output$penguinSummary <- DT::renderDT(
  expr = penStats,
  caption = "Descriptive Stats for Palmer Penguins",
  style = "bootstrap4",
  rownames = TRUE,
  autoHideNavigation = TRUE,
  options = list(
    responsive = TRUE,
    scrollX = TRUE,
    paging = FALSE, # Set to False for small tables
     columnDefs = list(
       list(className = 'dt-center',
            targets = 1:ncol(penStats))
    )
  )
)
```

12.4.4.2.1 Summary Data Table of Descriptive Statistics

12.4.4.2.2 Summary Data Table for Output Table While this example is for an ANOVA table, you can build from this for other output tables. If you store the output of any call as an object, you can then use the structure function, `str` to investigate the output. Ultimately, you need something that is either a matrix or a data frame.

```
library(palmerpenguins)
library(psych)
library(DT)
library(tibble)
library(rstatix)

# This bad practice but I'm going to pretend that all assumptions are met
penModel <- aov(body_mass_g ~ species*sex, data = penguins)

anovaPen <- round(anova(penModel), 3)
# Rounding to truncate decimals
```

```
# Make the Table
output$penguinAnova <- DT::renderDT(
  expr = anovaPen,
  caption = "(Classical) ANOVA Table for Palmer Pengins",
  style = "bootstrap4",
  rownames = TRUE,
  options = list(
    responsive = TRUE,
    scrollX = TRUE,
    paging = FALSE, # Set to False for small tables
     columnDefs = list(
       list(className = 'dt-center',
            targets = 1:ncol(anovaPen))
    )
  )
)
```

12.4.5 Static Images

Static image refers to any image you're using in your App which is not produced by `R`. These are usually PNG or JPG/JPEG files which you end up calling in the UI portion of your code.

Within your App's folder/directory, there needs to be a sub-folder/directory called `www`. This is the place where you'll need to place ALL static image files.

12.4.5.1 Adding an Image

To include the image in your App, you'll need to make use of the image tag, `img`. When you run your App, Shiny automatically knows to check the `www` folder any time the `img` tag gets called. Here is an example supposing that the check mark image for correct answers is in the app's `www` folder:

```
#[code omitted]
div(align = "right",
    img(src = "check.PNG",
        alt = "Success, you are correct",
        width = 25, #these are in pixels
        height = 25,
        ))
#[code omitted]
```

You'll notice that we've wrapped the `img` call in a `div` call. The `div` call allows us to specify that we want the image to be right aligned; you could also do left or center. You can replace the `div` call with the paragraph environment and include text on either side, effectively making your image part of the text.

```
#[code omitted]
p("Check your answer here -->",
  img(src = "check.PNG",
      alt = "Success, you are correct",
      width = 25, height = 25),
  "<-- Check your answer here"),
#[code omitted]
```

12.4.5.2 Sizing and Positioning Your Image

All image files have a native size that is part of that file. For instance, the check mark image is 270 x 250 pixels. However, we overrode that that sizing with the `width` and `height` arguments. How did we decide on 25 x 25? Honestly, through guessing and checking. You'll need to think about how you're using the image and let that guide your decision making. There is no one size fits all solution. While finding an optimal size and position for your image can take some time, seeing bad settings is pretty obvious. Feel free to reach out to Neil and Bob for assistance.

12.4.6 Alt Text

Any graphical element you include in your App **MUST** have an alternative (assistive) text description ("alt text"). This provides a short description of what is in the image or plot for users who are visual impaired. (Tables, when properly formatted will handle this automatically.)

Here are several resources worth checking out:

- WebAIM Alternative Text Guide
- Penn State's Image ALT Text Page
- W3C's ALT Text Decision Tree

12.4.6.1 Adding Alt Text to Static Images

In the prior section on static images, you saw exactly how to set the alt text; here is a generic example:

```
#[code omitted]
img(src = "yourImage.PNG",
    alt = "Short description of what's in the pic",
    width = 25, height = 25)
#[code omitted]
```

12.4.6.2 Adding Alt Text Graphs

At this point in time (6/10/2020), adding alt text to plots generated in `R` is not as easy as for static pictures. While there are some potential changes coming down the pipe, there is not a firm time line for the addition of appropriate arguments to the existing `render*` functions. Therefore we will need to put in some (hopefully) temporary measures.

In particular, we will use Accessible Rich Internet Applications (ARIA) to assist us in writing some labels that will stand in place of formal alt text. To do this, you will need to make use of the following code:

```
# [code omitted]
# In the UI section, in the appropriate tabItem
plotOutput(outputId = plotID) # Look for lines like this
# Code for adding the aria label
tags$script(HTML(
  "$(document).ready(function() {
```

```
  document.getElementById('plotId').setAttribute('aria-label',
  `General description of the plot`)
  })"
))
# [code omitted]
```

Important things to note:

1. Place the `tags$script(HTML(...))` code right after each instance of `plotOutput`.
2. Copy the above code as formatted
3. Change the two (2) pieces for each particular plot
    a. Replace `plotId` (keep the single quotation marks in the code)
    b. Replace `General description of the plot` (keep the single quotation marks in the code)

Part VI

Style Guide-Language

Chapter 13

Wording

This chapter focuses on the wording that we use within the BOAST Apps.

13.1 General Guidelines

When writing the content for your App, you will want to keep in mind that our apps have students as the primary audience. Thus, we need to make sure that we use language that is appropriate. Seek to use complete sentences that convey what you intend. Have someone else take a look at your content and then tell you what they believe the text to be saying. If what they say is consistent with what you intended, great. If not, then you need to revise your text.

DO NOT sacrifice clarity and precision/accuracy for conciseness/brevity. While we don't necessarily want a wall of text, there should still be some text to assist the user.

Since these apps are for *teaching and learning*, we need to use language that is accurate and supports students in constructing productive meanings. This means that we need to avoid sloppy language, re-enforcing problematic conceptions, and supporting fallacies. For example,

- Sloppy Language: Vagueness
  - BAD: "We want to explore how these averages differ."
  - NEUTRAL/FAIR: "We want to explore how these means differ."
  - GOOD: "We want to explore how the values of the *sample arithmetic mean* (*SAM*) varies between these groups."
- Sloppy Language: Discussing values of statistics
  - BAD: "The mean is 6."
  - GOOD: "The value of the *sample arithematic mean* for this data is 6 units/object."
- Problematic conceptions

    - BAD: "Probability is the likelihood of an event in relation to all possible events."
    - GOOD: "Probability is the long-run relative frequency for us seeing a particular data event given our assumptions. Likelihood is the long-run relative frequency of a set of assumptions being true given our collected data."
- Fallacies
    - BAD: "We're 95% confident that the true population proportion is between 0.35 and 0.45."
    - GOOD: "If we were to repeat the entire study infinitely many times, then 95% of the time we will make an interval that captures the true population proportion."

13.2 Popovers

The term "Popovers" refers to any number of different tools on websites that go by other names such as tool tips, rollovers, and hover text. In essence, this tool appears when the user places their cursor over (i.e., hover) or shifts the focus to a trigger object (typically user inputs or graphs). The text then pops up on the screen for the user to view. The function to create one of these in a Shiny app is `shinyBS::bsPopover`. While these can be powerful, they are often misused, leading to problems. For example, they can prevent the user from actually interacting with portions of your app when they appear.

Popovers are meant to provide short, simple clarifications; quick annotations which enrich the content that is already present. This type of text is meant to be temporary, only appearing for as long as the user is hovering/focusing on the trigger object. Thus, if you are putting information that is critical for a person to successfully use your App in a popover, you are using popovers **INCORRECTLY**.

Here are few additional sources for reading about Popovers/Tooltips:

- Tooltips in UI Design
- Tooltips: How to use this small but mighty UI pattern correctly

Restrict any usage of a popover to something short and non-vital for your App's user. If you do choose to use a popover, you will need to format the popover correctly. Be sure that your function call includes values for the following arguments:

- `id`: this needs to be the name of the object which will act as the trigger
- `title`: this will be a string that appears across the top of your popover; use verbs with an understood "you" (e.g., "Investigate!", "Remember", etc.)
- `content`: this will be the string that you want displayed; shorter is better.
- `placement`: this will control where the popover appears. Choose the

option (top, bottom, left, right) that works best for your space. Ensure that the placement does not cover any controls or other vital information.

The visual appearance of the popover will be control by the central BOAST CSS file.

13.3 Dealing with Differing Vocabularies

One of the most challenging aspects of Wording is the fact that we have to deal with the Jingle/Jangle problem.

13.3.1 The Jingle/Jangle Problem

A term "jingles" when people use that term to refer to two (or more) different concepts. This is also know as a term having lexical ambiguity. For instance, *random* jingles when people use the term to convey haphazardness, arbitrariness, and/or an attribute of process. (Note: the only the third option is statistically valid.)

On the other hand, a set of 2+ different terms "jangle" when they refer to the same concept. A good/classical example of this is skewness. Some people talk about left/right skewness, others negative/positive skewness, and others will talk about long left/long right tails. Each of these pairs refer to the same core concept, but evoke different mental images.

13.3.2 Option 0: Dictionary/Thesaurus

There are a variety of ways in which we could handle this approach. One thought is to build a Dictionary/Thesaurus for BOAST. While we could go down this route, this represents a considerable undertaking and might become a long term goal.

13.3.3 Option 1: Hover Text

The more immediate solution to differing vocabularies is for us to make use of hover text. This approach is appealing in that the user doesn't have to leave your App (like going to a dictionary/thesaurus) and the content of these tips is not critical to using your App. That is to say, the information is available for those who might need a quick reminder but does not take up permanent screen space.

To use this tool you'll need to make sure that you install and load the `tippy` package.

```
install.packages("tippy")
```

Suppose that we want to add the hover text to the the word "positive" in the following sentence:

> A positively skewed histogram will hoave potential outliers that are larger than the main modal clump(s).

We would need to do the following in our `app.R` (or `ui.R`) file:

```r
# Required Library
library(tippy)
# In UI Section
#[code omitted]
dashboardBody(
  #[code omitted]
  tabItem(
    tabName = "Overview",
    h1("Exploring Skewness"),
    p("A ",
      tippy::tippy(text = "positively skewed",
                   tooltip = "Sometimes called 'long right tail' or 'right skewed'",
                   arrow = TRUE, placement = "auto"),
      " histogram will have potential outliers that are larger than the main modal clu
  #[code omitted]
))
```

There are several things to notice in the example:

1. The `tippy` call is part of the paragraph environment. If the text is going to be part of a list, then the `tippy` call should be part of a list item.
2. There are four (4) required arguments:
   a. `text`—the words that will be part of the page

   b. `tooltip`—the words that will appear/disappear when the user hovers/focuses on the `text`
   c. `arrow = TRUE`—this creates an arrow from the `tooltip` to the `text`. Make sure to set this as `TRUE`
   d. `placement`—controls where the the `tooltip` appears in relation to the `text`. While there are multiple values you could use here, we recommend using `auto` to allow the App to determine the best position. If you want to override this, then you should use values of `top`(shows above) or`bottom` (shows below).
3. Be sure to include spaces around text that appears before and after the `tippy` element.
4. Don't forget to put commas between the text and the `tippy` element. These won't appear in your App but allow R to see that there are multiple elements.

13.3.3.1 Choosing Terms-Jangle

If you are going to make use of hover text to combat a jangle problem, you are going to have to make a decision about what words/phrase will be part of the app (i.e., the `text`) and which words/phrases will be part of the hover text (i.e., the `tooltip`).

You should make this decision in conjunction with a faculty member. Our recommendations are to use the word or phrase which:

1. Best supports students in building productive meanings
2. Best supports students in seeing coherence between a variety of concepts

Appealing to "tradition" or "this what most people do" IS NOT a valid justification. Again, these apps are to support students in building their understandings, we must do better.

In the `tippy` example above, the ordering of terms is:

1. positively skewed
2. long right tail
3. right skewed

This ordering reflects the ordering from most productive and coherent to least. "Positively skewed" works regardless of the orientation of the histogram (see Figure 13.1) as well as directly connecting to the statistic *sample skewness*. The later two only make sense in one orientation and do not connect to *sample skewness*. "Long right tail" is preferable to "right skewed" as this phrasing helps students avoid the common belief that the position term (right/left) is about where the bulk of the observations are.

13.3.3.2 Choosing Terms-Jingle

We will always use the statistical/probabilistic meaning for a term, never the colloquial/everyday/non-technical meaning(s).

13.3.4 Option 2: Entry on the Prerequisites Tab/Page

Another option that you could do for both the jingle and jangle problems is to add an entry on the Prerequisites page. If you only have a jangle problem, you can use the Hover Text option.

Option 2 can be combined with Option 1.

13.4 Footnotes

We recommend avoiding footnotes in favor of Hover Text.

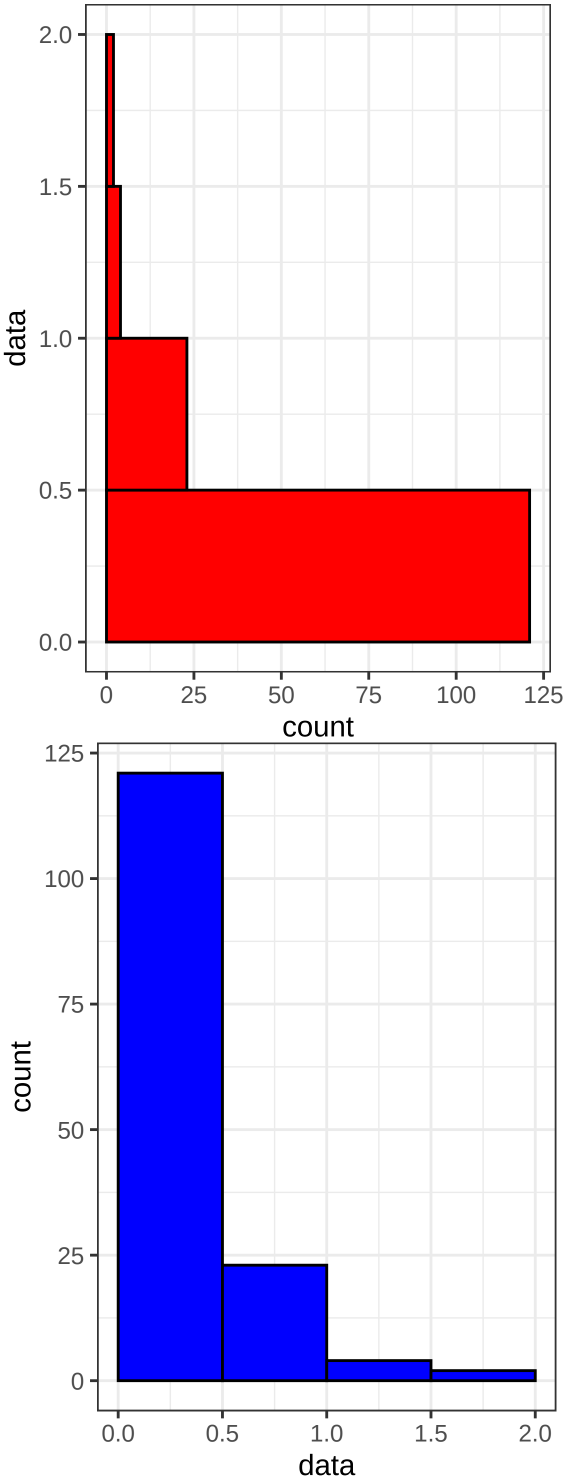


Figure 13.1: Same Data, Two Orientations

Chapter 14

Documentation

These apps are the product of your hard work and are part of your academic record. Thus, you need to adhere to Penn State's Academic Integrity Policy. This is especially important as we are making the apps available through a Creative Commons Attribution-NonCommerical-ShareAlike 4.0 International license (CC BY-NC-SA 4.0). If you have used code, pictures, data, or other materials from outside of the BOAST team, you **MUST** give proper credit. These references will then be included on the App's References Tab.

14.1 References

All apps will need a References Tab. This is where you'll place all references for your App, including R packages, borrowed code, data sources, images, etc. This in addition to the Acknowledgments.

NOTE: listing something in the Acknowledgments DOES NOT waive this requirement.

We will make use of the American Statistical Association's reference style. Please check with Neil, Matt, and Dennis for getting assistance. You can see this style in action, with the Reference Tab example of Chapter 8. Here is a starting code block for you to use:

```r
#[omitted code]
tabItem(
  tabName = "refs",
  withMathJax(),
  h2("References"),
  p(class = "hangingindent",
    "reference 1-alphabetically"),
  p(class = "hangingindent",
    "reference 2-alphabetically"),
  # Repeat as needed
)
```

Notice the use of `class = "hangingindent"`. You must include this with each reference as this will ensure the proper styling of your references.

If you need assistance with this section, please talk to Neil.

14.2 Plagiarism

You MAY NOT use blocks of code you've found online without giving proper attribution.

There is a difference between looking at example code online to see how to do something and copying that code directly. The former is permissible, the later is plagiarism.

- If you want to use someone else's code "as is" (without any changes), you should reach out to the author for permission first.
- If you use someone else's code and make modifications, you need to give credit to where you got the code, and potentially ask for permission.

You will need to place citations in ***two*** places: in the References Tab and in your code. You might want to also consider adding an acknowledgement to the Overview tab.

14.2.1 Reference Tab

Use the following format:

> Author. (Date), Title of program/source code, [type of code]. Available from < URL >.

For example,

> Hatfield, N. J. (2017), First day activity, Netlogo. Available from https://neilhatfield.github.io/statApps/Day1Activity.html.

14.2.2 In Code

Use the following format in your code to cite where you got the code from.

```
#----------------------------------------------------------------------------
#  Title: <title of program/source code>
#  Author: <author(s) names>
#  Date: <date>
#  Code version: <code version>
#  Availability: <where it's located>
```

```
#-------------------------------------------------------------------------------
# [borrowed code then follows]
# ...
# [last line of borrowed code]
#End of <author>'s code----------------------------------------------------------
```

14.3 R Packages

If you made use of any packages in R, then you will need to add these to the Reference tab. Fortunately, there is a built-in tool that will help you: the `citation` function. In R (RStudio) simply type `citation("packageName")` and you'll get the appropriate citation information for the package you used. For example, `citation("shinydashboard")` and `citation("plyr")` will give the information needed for the following citations:

> Chang, W. and Borges Ribeio, B. (2018), shinydashboard: Create dashboards with 'Shiny', R Package. Available from https://CRAN.R-project.org/package=shinydashboard
>
> Wickham, H. (2011), "The Split-apply-combine strategy for data analysis". Journal of Statistical Software, 40, pp. 1-29. Available at http://www.jstatsoft.org/v40/i01/.

Notice, that the format of the R package will depend on whether there is an article published for the package. The `shinydashboard` package is not associated with an article while the `plyr` package is associated with Wickham's article.

14.4 Graphics

Pictures, drawings, photographs, images, etc. are typically copyrighted. When you're selecting images, make sure that the images are Open Source/Copyright Free/Royalty Free/Public Domain. Additionally, include a reference to where the pictures came from in the Overview Page. The basic format to use is:

> LastName, First Initial. (Year), Title of artwork. Retrieved from < URL > (if available).

14.5 Data

If you are using any data files, you need to attribute where those files are coming from in the References tab. You might also want to add an acknowledgment on the Overview tab. A suggested format to use is:

> Author/Rightsholder. (Year), Title of data set, [Description of form], Location: Name of producer.

> Author/Rightsholder. (Year), Title of data set, [Description of form].
> Available at http://www.url.com

If you (or someone else) had to sign some type of agreement to access the data, we must examine the agreement before you make your App publicly accessible. Just because you got access to the data does not mean you have the right to share the data.

Part VII

Accessibility and Mobile Devices

Chapter 15

Accessibility

We need to make sure that our Apps are accessible. If you have been adhering to the style guide, your App should be in a decent position. When you're ready to test the accessibility of your App, you'll need to deploy the App to a sever and then use the WAVE Web Accessibility Evaluation Tool. Enter the URL of your App in the noted box to run an evaluation. See what accessibility issues your App has and then address them.

See also:

- Accessibility and Usability at Penn State
- Accessibility Statement
- How to Meet Web Content Accessibility Guidelines
- 7 Things Every Designer Needs to Know about Accessibility

15.1 Making Your App Accessible

The best way that you can make your App accessible is to adhere to this Style Guide as closely as possible. Sadly, the Shiny framework is woefully behind the times for making truly accessible apps. We have tried to strike a balance so that you won't be inundated with significant burdens for accessibility. By following this Style Guide, you'll place your App in the best position possible.

As we figure out and develop new tools to help you improve the accessibility of your App, we might ask you to return to your App and make updates.

15.2 Checking Accessibility

There are a couple of different phases to testing an app for Accessibility: Testing, Reading a Report, and Addressing Issues.

15.2.1 Testing Accessibility

We highly recommend using the WAVE Web Accessibility Evaluation Tool. If you use Firefox or Chrome, we recommend that you install the WAVE Browser Extension as this will give you a quick way to test your App's accessibility.

Depending on whether you're checking an app that is already in BOAST or a revision/new app, you'll need to access the WAVE tool differently.

15.2.1.1 Testing an App in BOAST

To check an App that is currently linked in BOAST:

1. Go to WAVE Web Accessibility Evaluation Tool
2. In a new tab, launch the app from BOAST
3. Copy the app's URL from the address bar of your browser
4. Paste the URL in the Web page address field of WAVE

15.2.1.2 Testing a Revision/New App

To check an App that you are currently revising and/or developing (i.e., the App has *not* been formally added to BOAST):

1. Make sure you install the WAVE Browser Extension for Firefox or Chrome.
2. Click the Run App button in RStudio
3. In the resulting window, click the Open in Browser button located along the top edge
4. Activate the WAVE Browser Extension

To help make the report more friendly with your browser, we also recommend you install the Stylus Add-on for Firefox or the Stylus Add-on for Chrome. Once you've installed Stylus, you'll want to

1. Create a new Style in Stylus

2. Add the following code:

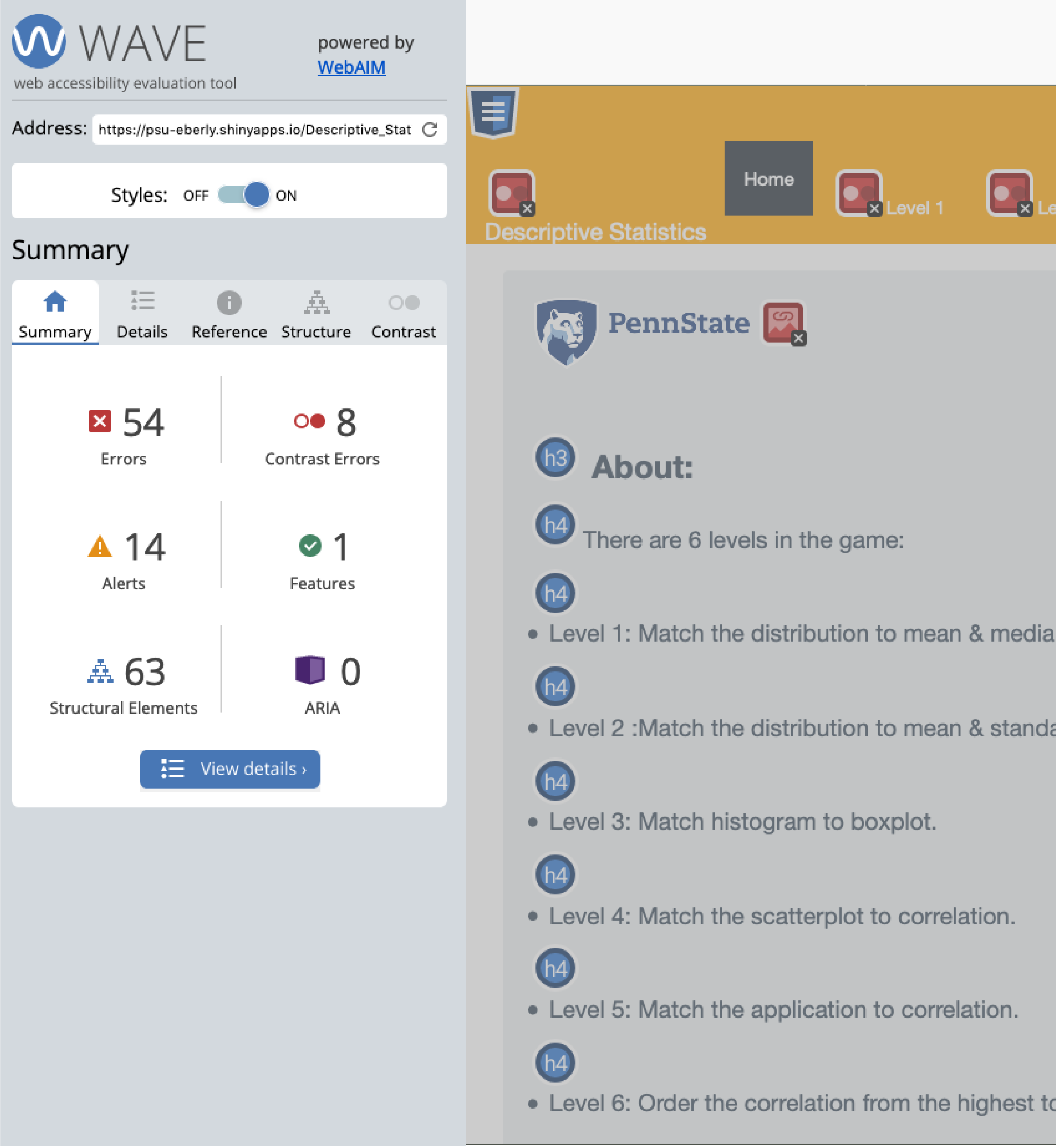
   ```
   body.ng-scope {
     position: relative
   }
   ```

3. Save and close the Stylus tab

This will move the the WAVE summary bar further to the left. If you have questions, please reach out to Neil and Bob.

15.2.2 Reading a WAVE Report

Figure 15.1 shows the WAVE report for the Descriptive Statistics App. There are 54 errors including missing alternative text, empty headings. There are also 8 contrast errors, 14 alerts (skipping heading levels),



Address: https://psu-eberly.shinyapps.io/Descriptive_Stat

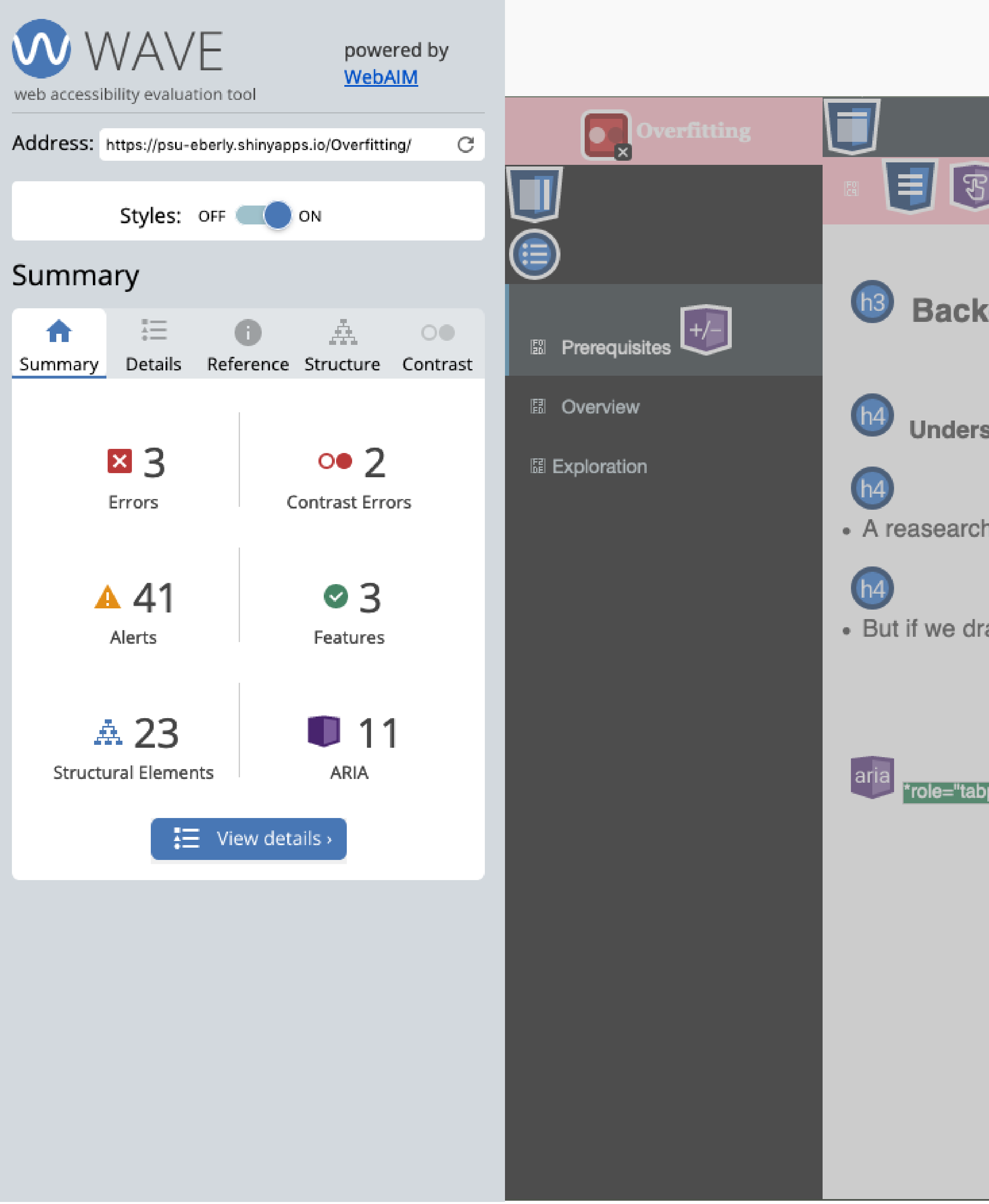
Styles: OFF ● ON

About:

There are 6 levels in the game:

- Level 1: Match the distribution to mean & media
- Level 2 :Match the distribution to mean & standa
- Level 3: Match histogram to boxplot.
- Level 4: Match the scatterplot to correlation.
- Level 5: Match the application to correlation.
- Level 6: Order the correlation from the highest to

Figure 15.1: WAVE Report for Descriptive Statistics App



Address: https://psu-eberly.shinyapps.io/Overfitting/

Styles: OFF ON

Summary

3 Errors

2 Contrast Errors

41 Alerts

3 Features

23 Structural Elements

11 ARIA

Figure 15.2: WAVE Report for Overfitting App

While there are fewer errors for the Overfitting App (Figure 15.2), there are a lot of alerts on this app including skipping heading levels and small text.

The six categories of items in the WAVE report reflect the areas in which you're going to work on identifying issues. The two most critical issues are Errors and Contrast Errors. While the goal would be to have 0 for both, we do anticipate a few issues that we will have to leave unresolved. The third category that you need to attend to is the Alerts group. Again, we would like this number to be 0. The Features, Structural Elements, and ARIA categories are informational and you don't necessarily need to worry about them (they don't signify problems).

After running WAVE and getting the report, click on the View details button to see the list of all items found to see details. Additionally, you'll be able to click on items in the Details to see which things in your App are being flagged.

15.2.3 Addressing Issues

By looking at the details you'll be able to identify what issues you need to address. We fully expect that some issues (e.g., Error: missing alt text, Alert: missing first level heading) are things that you can directly fix. However, we also anticipate that there are some issues that you might not be able to fix (e.g., Error: Document language missing) without assistance.

We recommend that you document all errors and alerts that you've yet to resolve with you make your Pull Request. The decision about whether to leave an Accessibility Error or Alert alone/un-addressed has to be made by a faculty member (e.g., Neil, Dennis, Matt, or Bob), not any student.

By using the `boastApp` function and adhering to this Style Guide, your App should be in a good initial position. After running your App through WAVE, we can work with you address any changes that might be necessary.

Chapter 16

Mobile Friendliness

We want our apps to work well with mobile devices. Thus, when you get to the point where the majority of bugs have been fixed, you need to check how mobile friendly your App is. If you have used `boastApp` and/or the `boast.CSS` file, along with the practices laid out earlier, then you should be well on your way to being mobile friendly.

You can check your App in two ways:

1. Test your App out on a variety of mobile devices.
2. Make use of a browser's ability to mimic devices. To do this, launch your App in a browser, then enable one of the following:
   - Chrome: Device Mode
   - Firefox: Responsive Design Mode
   - Microsoft Edge: Device Emulation
   - Safari: Responsive Design Mode

Look for any issues that you might be able to address before you hand off your App for others to play around with. Assign a **Mobile Friendliness Rating** to your App on a scale from 1 to 5.

1. Not functional
2. Functional – Very awkward
3. Functional – Okay if no big screen available (multiple issues)
4. Usable in a small class setting (single issue)
5. Readily usable

Part VIII

Final Thoughts

Chapter 17

Additional Tools

Here are a few additional tools that can help you with App development:

- lintr - Checks adherence to a given style, syntax errors, and possible semantic issues.
- styler - Format R code according to a style guide.
- funchir - stale package check

R Code

```r
install.packages(c("funchir", "lintr", "styler"))
```