An illustrated guide to adding data, managing the database, running analyses, and more

KINOPEDIA

Official User Manual (v1.0)

Joyce, Alex William

This document will explain how to use and maintain the Kinopedia code base. The current code can be found on the CDRL’s GitHub page.

Contents:

1. General Data Structure

1-1 Kinopedia Directory	2
1-2 Major Kinopedia Features	3
1-3 S4 Class Structure	3

2. Database Structure

2-1 Kinase Runs	6
2-2 Kinase Data Objects	7
2-3 Database Integration	8
2-4 Adding New Kinase Datasets	9

3. Rshiny Structure

3-1 Modules	12
3-2 Primer on Reactivity	13

4. Specific Module Features

4-1 The Lookup Module	16
4-2 The Comparison Module	18
4-3 The EnrichR Module	20

5. Function Guide

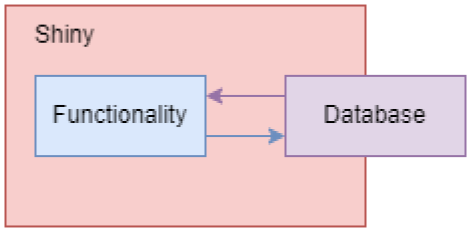
5-1 Normal Functions	23
5-2 Generics	34

1. General Data Structure

Kinopedia contains several functionally different sections that constitute the basis of the program. These sections are:

- Database; files related to the kinases contained within the program.
- Shiny; components related to the application itself. This includes all of the modules that facilitate the data analysis.
- Functionality; components related to application functions such as heatmaps, plots, and statistical analyses.

Understanding how these three components interact with one another is paramount to figuring out how the program itself works. The database is largely separate from the rest of the program, existing externally and being called when needed. The functionality is closely linked to the Shiny components, which can be thought of as a 'scaffold' in which the functionality is held.



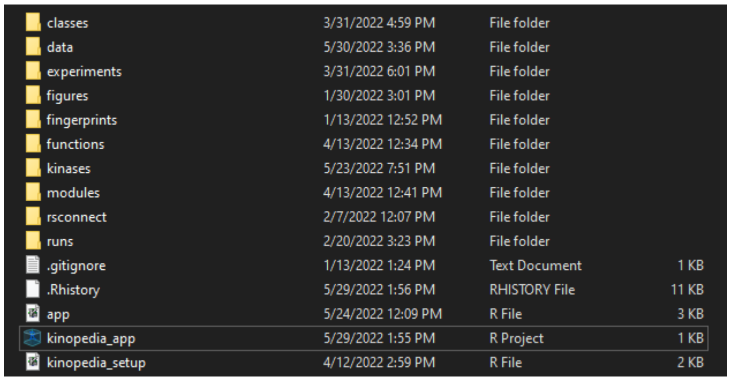
1-1 Kinopedia directory

The main directory of Kinopedia contains the following folders:

- Classes: script files for S4 class definitions
- Data: setup files needed for Kinopedia to run
- Experiments: experimental data
- Fingerprints: fingerprint data
- Functions: script files for Kinopedia functions
- Kinases: recombinant kinase data
- Modules: scripts related to the modules for the Rshiny component
- Rsconnect: used to connect the program to shinyapps.io
- Runs: contains data related to each 'run' (set of chips run on the machine)

There are also several important files located within the main directory:

- .gitignore/.Rhistory: files that can be ignored
- app.R: primary file used to launch the Shiny app
- kinopedia_app.Rproj: Rproject file- used to modify the program in Rstudio
- kinopedia_setup.R: script used by app.R to load all of the essential components from the data, modules, and function folders.



classes	3/31/2022 4:59 PM	File folder	
data	5/30/2022 3:36 PM	File folder	
experiments	3/31/2022 6:01 PM	File folder	
figures	1/30/2022 3:01 PM	File folder	
fingerprints	1/13/2022 12:52 PM	File folder	
functions	4/13/2022 12:34 PM	File folder	
kinases	5/23/2022 7:51 PM	File folder	
modules	4/13/2022 12:41 PM	File folder	
rsconnect	2/7/2022 12:07 PM	File folder	
runs	2/20/2022 3:23 PM	File folder	
.gitignore	1/13/2022 1:24 PM	Text Document	1 KB
.Rhistory	5/29/2022 1:56 PM	RHISTORY File	11 KB
app	5/24/2022 12:09 PM	R File	3 KB
kinopedia_app	5/29/2022 1:55 PM	R Project	1 KB
kinopedia_setup	4/12/2022 2:59 PM	R File	2 KB

1-2 Major Kinopedia Features

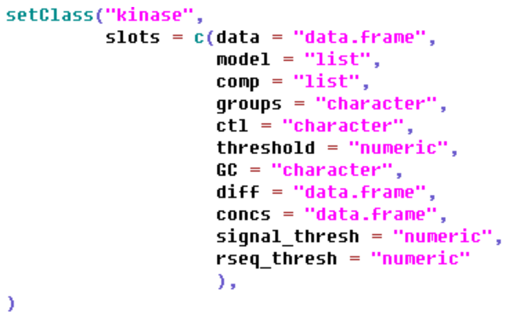
Kinopedia has several different features that can be used to analyze different types of kinase data. These features include:

- Viewing interactive heatmaps through Plotly
- Browsing the database
- Comparing different datasets
  - Generating heatmaps of log fold-change values
  - Performing correlation analysis
- Pathway analysis

These features will be described in greater detail in the following sections.

1-3 S4 Class Structure

Kinopedia is based around S4 classes, which can be a bit of a confusing concept to understand. As these classes encapsulate the majority of the data within the Kinopedia database, it is important to have a basic understanding of how this system works before modifying the database. Put simply, S4 classes act as a form of extended list that allows a single object to have multiple components. For instance, the kinase class contains the model, data, concentration, and general information regarding a specific kinase; this information can then be accessed by ‘methods’, which will be described shortly.



```
setClass("kinase",
         slots = c(data = "data.frame",
                   model = "list",
                   comp = "list",
                   groups = "character",
                   ctl = "character",
                   threshold = "numeric",
                   GC = "character",
                   diff = "data.frame",
                   concs = "data.frame",
                   signal_thresh = "numeric",
                   rseq_thresh = "numeric"
                   ),
)
```

When new data is added, the *build_kinase( )* function constructs a new kinase object by filling the necessary slots. Due to differences in data, there are two instances of this function: *build_kinase_conc( )* and *build_kianse_rep( )*; despite this, they both work the same.

```
build_kinase_conc <- function(data, model, concs){
  data <- data.frame(data)
  kin <- new("kinase",
             data = data,
             model = model,
             concs = concs,
             comp = list(comp1 = c("High", "CTL"),
                         comp2 = c("Medium", "CTL"),
                         comp3 = c("Low", "CTL")),
             groups = c("High", "Medium", "Low"),
             ctl = "CTL",
             threshold = 0.1,
             GC = c("High", "CTL"),
             signal_thresh = 0,
             rseq_thresh = 0
             )
  kin
}
```

```
build_kinase_rep <- function(data, model, concs){
  data <- data.frame(data)
  kin <- new("kinase",
             data = data,
             model = model,
             concs = concs,
             comp = list(comp1 = c("Replicate_1", "CTL"),
                         comp2 = c("Replicate_2", "CTL"),
                         comp3 = c("Replicate_3", "CTL")),
             groups = c("Replicate_1", "Replicate_2", "Replicate_3"),
             ctl = "CTL",
             threshold = 0.1,
             GC = c("Replicate_1", "CTL"),
             signal_thresh = 0,
             rseq_thresh = 0
             )
  kin
}
```

Understanding how this translates into the main code is a bit more complicated. Knowing that these classes act as lists, it is possible to access these components through the '@' notation. For example, if a function requires a model as an argument, the notation 'kinase@model' can be used. To further simplify this, we can use generics and methods. In the 'classes' folder, there is a script called 'generics.R'. Opening it, we can see generics such as:

```
setGeneric("lfc_table",
           function(x) standardGeneric("lfc_table"),
           signature = 'x'
           )
```

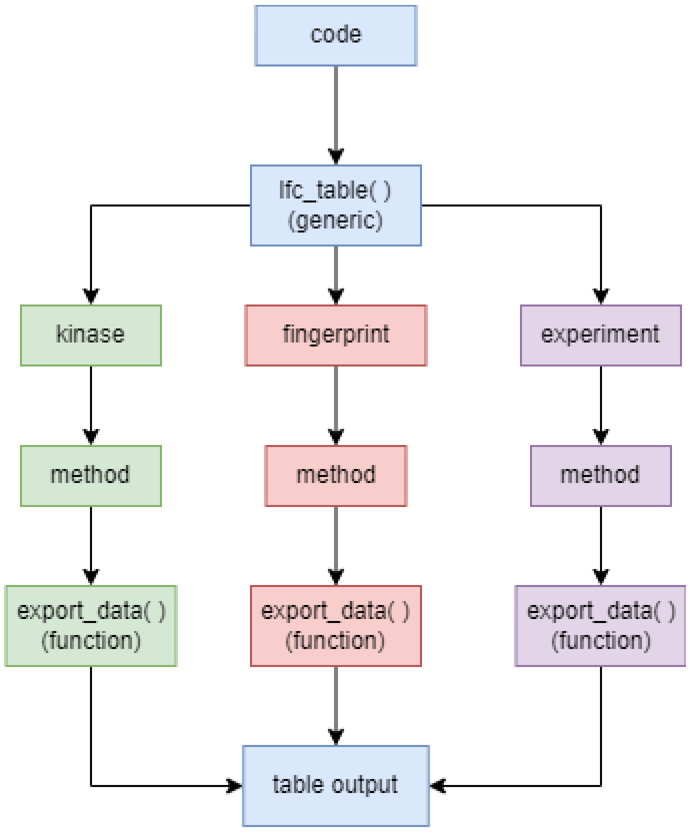
This generic is for generating the LFC tables used in many of Kinopedia's functions. Moving back to the kinase_class.R file, something similar can be seen further down in the script:

```
setMethod("lfc_table", "kinase", function(x){
                  export_data(model = x@model,
                              data = x@data,
                              threshold = x@threshold,
                              groups = x@GC,
                              sig_thresh = x@signal_thresh,
                              rseq_thresh = x@rseq_thresh)
          })
```

This is the corresponding method for the lfc_table( ) generic. Within this method, we can see a function called 'export_data( )' which is the function that actually generates the table. Although this appears complex on the surface, it is rather simple in practice. The final piece of this puzzle is to see how this code is implemented in the program itself. The screenshot below shows an instance of lfc_table( ) within the recombinant module:

```
diffDF <- reactive({
  lfc_table(run$userInput)
})
```

Here, 'run$userInput' is a kinase-class object (think 'ABL2'; the reason for this is due to reactive programing within Rshiny, which is described later). In essence, lfc_table( ) is a function that can accept any of the three data types and generate a table of log fold-change values from them. It does this by recognizing the argument as a kinase-class object, then calls the appropriate method (method dispatch). The job of the method is to look at the slots contained within the kinase class then translate these slots to arguments used by the 'export_data( )' function. This is summarized in the following chart:



2. Database Structure

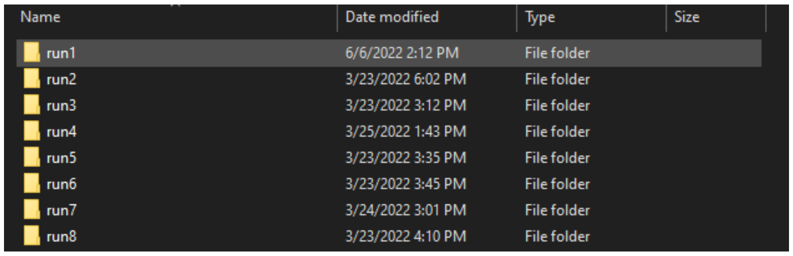
The database is at the heart of Kinopedia; it contains all of the built-in data related to protein kinases. In order to facilitate a streamlined experience, the Kinopedia database has a specific format that must be followed. Roughly, the database can be divided into two sections: the ‘run’ section, which contains all of the instructions necessary to generate new files, and the ‘data’ section, which holds the actual kinase data.

Based on the S4 structure of Kinopedia, each protein kinase (recombinant, experimental, or fingerprint) can be thought of as an ‘object’. Programmatically, this was described as being akin to a type of list, however, in terms of physical storage, we can think of each kinase as a folder (directory) of several files. Over the next two sections, this schema will be described in greater detail using recombinant data as an example.

2-1 Kinase Runs

The raw data for the kinome array is normally found in the context of a *run*. A run is a collection of three or more chips identified by a unique barcode. Because each run can contain multiple different kinases, it is important that pre-processing steps are put into place to separate each unique kinase in the run.

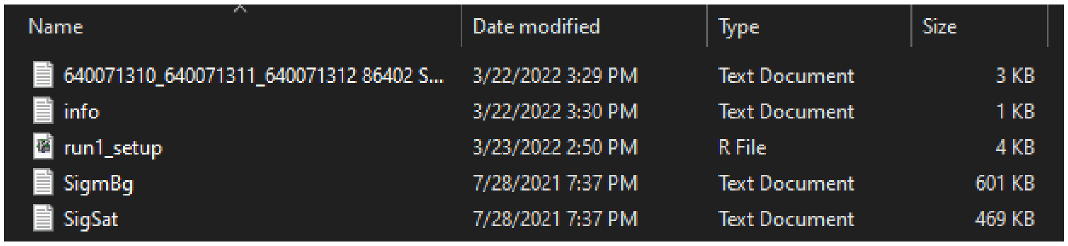
Information on each run is contained with the ‘runs’ folder in the main directory.



Name	Date modified	Type	Size
run1	6/6/2022 2:12 PM	File folder	
run2	3/23/2022 6:02 PM	File folder	
run3	3/23/2022 3:12 PM	File folder	
run4	3/25/2022 1:43 PM	File folder	
run5	3/23/2022 3:35 PM	File folder	
run6	3/23/2022 3:45 PM	File folder	
run7	3/24/2022 3:01 PM	File folder	
run8	3/23/2022 4:10 PM	File folder	

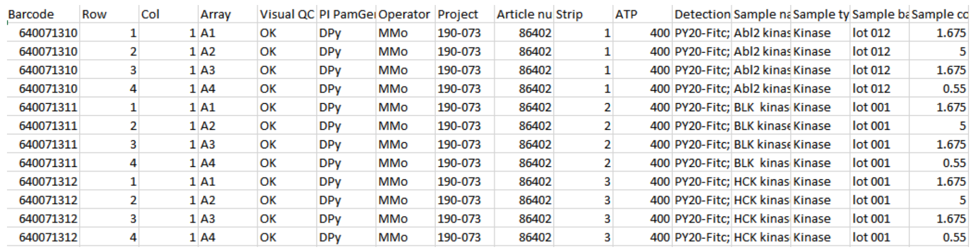
Within each of these numbered folders is a set of several files:

- Info: .txt file that lists all of the kinases and their associated barcodes
- Setup: R script that provides the code responsible for making the files
- SigmBg/SigSat: KRSA files that contain the actual run data
- Sample Annotation: additional metadata used in the preprocessing workflow.



Name	Date modified	Type	Size
640071310_640071311_640071312 86402 S...	3/22/2022 3:29 PM	Text Document	3 KB
info	3/22/2022 3:30 PM	Text Document	1 KB
run1_setup	3/23/2022 2:50 PM	R File	4 KB
SigmBg	7/28/2021 7:37 PM	Text Document	601 KB
SigSat	7/28/2021 7:37 PM	Text Document	469 KB

Most of the Setup.R script file is taken from the KRSA .Rmd template and can be run as-is, however, there is an important caveat to consider. To streamline the process of loading new data, a standardized naming system was implemented. This involves the use of the ‘name_eval()’ function which replaces the sample names with ‘High, Medium, Low, and CTL’ for concentration series or ‘Rep1, Rep2, Rep3, and CTL’ for replicates. Because not every set of data contains the same sample order, it is important to make sure that the ‘vec’ argument in the script file is correct. Sample order can be identified from the sample_annotation.txt file as shown below:



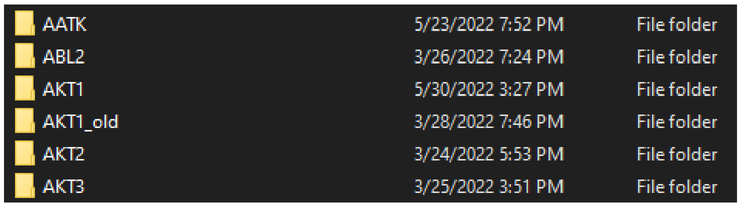
Barcode	Row	Col	Array	Visual QC	PI PamGe	Operator	Project	Article nu	Strip	ATP	Detection	Sample na	Sample ty	Sample ba	Sample cc
640071310	1	1	A1	OK	DPy	MMo	190-073	86402	1	400	PY20-Fitc;	Abl2 kinas	Kinase	lot 012	1.675
640071310	2	1	A2	OK	DPy	MMo	190-073	86402	1	400	PY20-Fitc;	Abl2 kinas	Kinase	lot 012	5
640071310	3	1	A3	OK	DPy	MMo	190-073	86402	1	400	PY20-Fitc;	Abl2 kinas	Kinase	lot 012	1.675
640071310	4	1	A4	OK	DPy	MMo	190-073	86402	1	400	PY20-Fitc;	Abl2 kinas	Kinase	lot 012	0.55
640071311	1	1	A1	OK	DPy	MMo	190-073	86402	2	400	PY20-Fitc;	BLK kinas	Kinase	lot 001	1.675
640071311	2	1	A2	OK	DPy	MMo	190-073	86402	2	400	PY20-Fitc;	BLK kinase	Kinase	lot 001	5
640071311	3	1	A3	OK	DPy	MMo	190-073	86402	2	400	PY20-Fitc;	BLK kinase	Kinase	lot 001	1.675
640071311	4	1	A4	OK	DPy	MMo	190-073	86402	2	400	PY20-Fitc;	BLK kinas	Kinase	lot 001	0.55
640071312	1	1	A1	OK	DPy	MMo	190-073	86402	3	400	PY20-Fitc;	HCK kinas	Kinase	lot 001	1.675
640071312	2	1	A2	OK	DPy	MMo	190-073	86402	3	400	PY20-Fitc;	HCK kinas	Kinase	lot 001	5
640071312	3	1	A3	OK	DPy	MMo	190-073	86402	3	400	PY20-Fitc;	HCK kinas	Kinase	lot 001	1.675
640071312	4	1	A4	OK	DPy	MMo	190-073	86402	3	400	PY20-Fitc;	HCK kinas	Kinase	lot 001	0.55

What the name_eval() function does is it looks at each value of the array column (A1 through A4) and replaces the sample name that corresponds to that array with the value of the matching number in ‘vec’. Essentially, if the control sample in the experiment is located at array A1, the first item in ‘vec’ should be ‘CTL’.

The rest of the setup.R script is straightforward; its main purpose is to generate the model, data, and conc (concentration) files, which are all the primary components of the kinase data objects discussed in the next section.

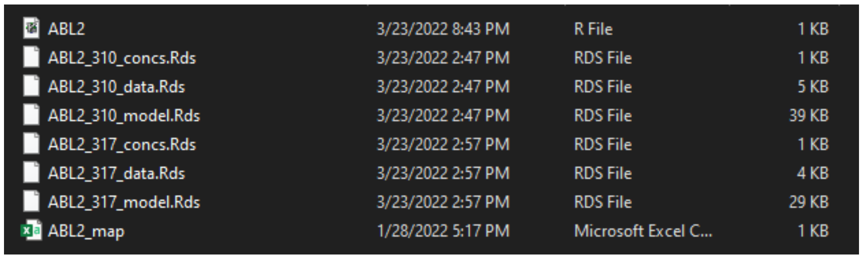
2-2 Kinase Data Objects

Processed kinase data is contained within several different directories depending on the type of data. In the case of recombinant kinase data, the data is contained within the ‘kinase’ directory.



```
AATK      5/23/2022 7:52 PM   File folder
ABL2      3/26/2022 7:24 PM   File folder
AKT1      5/30/2022 3:27 PM   File folder
AKT1_old  3/28/2022 7:46 PM   File folder
AKT2      3/24/2022 5:53 PM   File folder
AKT3      3/25/2022 3:51 PM   File folder
```

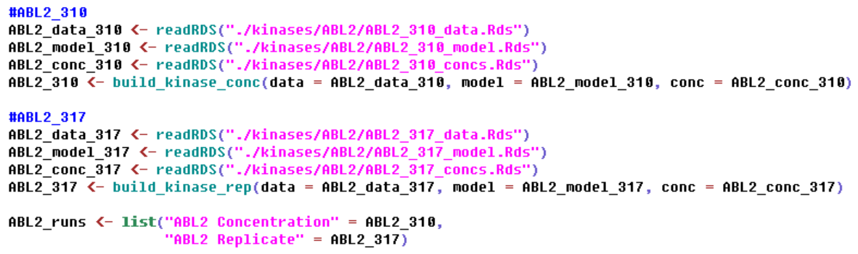
In this directory, we can see several named folders. Each of these folders contains various files that comprise the kinase object itself.



```
ABL2                  3/23/2022 8:43 PM   R File                1 KB
ABL2_310_concs.Rds    3/23/2022 2:47 PM   RDS File              1 KB
ABL2_310_data.Rds     3/23/2022 2:47 PM   RDS File              5 KB
ABL2_310_model.Rds    3/23/2022 2:47 PM   RDS File             39 KB
ABL2_317_concs.Rds    3/23/2022 2:57 PM   RDS File              1 KB
ABL2_317_data.Rds     3/23/2022 2:57 PM   RDS File              4 KB
ABL2_317_model.Rds    3/23/2022 2:57 PM   RDS File             29 KB
ABL2_map              1/28/2022 5:17 PM   Microsoft Excel C...  1 KB
```

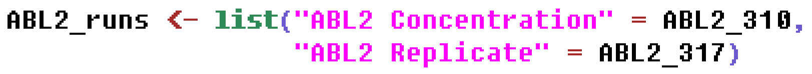
In the ABL2 folder, for example, two distinct sets of concentration, data, and model files can be seen along with a map.csv file. The reason that there are two sets of data for the same kinase is the fact that there were two runs performed on the same kinase set (ABL2, BLK, and HCK) under different conditions. The first run consisted of variable amounts of protein within each well (concentration series) while the second run used the same amount of protein (replicate series). Different runs of the same kinase are distinguished by affixing the last three digits of the barcode to the kinase's name and specifying the component, i.e, 'ABL2_310_model.Rds'.

Examining the ABL2.R file shows how these components are brought together to create the finalized kinase object. Understanding how the data is assembled within Kinopedia is crucial to understanding how the shiny and functional components of the app operate as well as to update and maintain the database.



```r
#ABL2_310
ABL2_data_310 <- readRDS("./kinases/ABL2/ABL2_310_data.Rds")
ABL2_model_310 <- readRDS("./kinases/ABL2/ABL2_310_model.Rds")
ABL2_conc_310 <- readRDS("./kinases/ABL2/ABL2_310_concs.Rds")
ABL2_310 <- build_kinase_conc(data = ABL2_data_310, model = ABL2_model_310, conc = ABL2_conc_310)

#ABL2_317
ABL2_data_317 <- readRDS("./kinases/ABL2/ABL2_317_data.Rds")
ABL2_model_317 <- readRDS("./kinases/ABL2/ABL2_317_model.Rds")
ABL2_conc_317 <- readRDS("./kinases/ABL2/ABL2_317_concs.Rds")
ABL2_317 <- build_kinase_rep(data = ABL2_data_317, model = ABL2_model_317, conc = ABL2_conc_317)

ABL2_runs <- list("ABL2 Concentration" = ABL2_310,
                  "ABL2 Replicate" = ABL2_317)
```

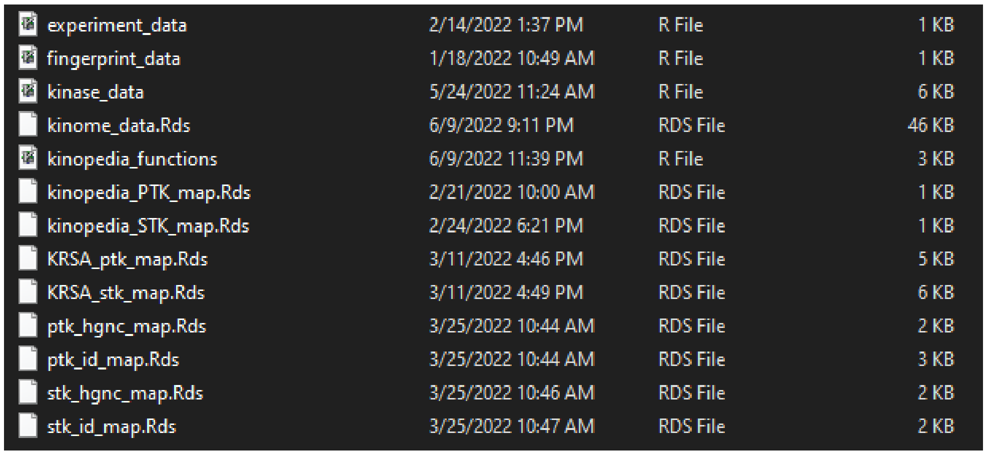
Here, it can be seen that two separate kinase data objects are being built from the constructors 'build_kinase_rep( )' and 'build_kinase_conc( )'. These objects are then placed into a list which can be accessed by the program itself.



```r
ABL2_runs <- list("ABL2 Concentration" = ABL2_310,
                  "ABL2 Replicate" = ABL2_317)
```

2-3 Database Integration

The way that Kinopedia reads the kinase data is rather straightforward. Most of the information regarding how the data is loaded into Kinopedia is contained within the 'data' directory.



```
experiment_data          2/14/2022 1:37 PM     R File      1 KB
fingerprint_data         1/18/2022 10:49 AM    R File      1 KB
kinase_data              5/24/2022 11:24 AM    R File      6 KB
kinome_data.Rds          6/9/2022 9:11 PM      RDS File   46 KB
kinopedia_functions      6/9/2022 11:39 PM     R File      3 KB
kinopedia_PTK_map.Rds    2/21/2022 10:00 AM    RDS File    1 KB
kinopedia_STK_map.Rds    2/24/2022 6:21 PM     RDS File    1 KB
KRSA_ptk_map.Rds         3/11/2022 4:46 PM     RDS File    5 KB
KRSA_stk_map.Rds         3/11/2022 4:49 PM     RDS File    6 KB
ptk_hgnc_map.Rds         3/25/2022 10:44 AM    RDS File    2 KB
ptk_id_map.Rds           3/25/2022 10:44 AM    RDS File    3 KB
stk_hgnc_map.Rds         3/25/2022 10:46 AM    RDS File    2 KB
stk_id_map.Rds           3/25/2022 10:47 AM    RDS File    2 KB
```

Inside this directory are several script (.R) and data (.Rds) files. The script file 'kinase_data.R' contains several paths to the scripts from the 'kinase' directory.

```
#ABL2
source("./kinases/ABL2/ABL2.R")

#AKT1
source("./kinases/AKT1/AKT1.R")

#AKT2
source("./kinases/AKT2/AKT2.R")

#AKT3
source("./kinases/AKT3/AKT3.R")
```

At the bottom of this script are three lists. The first two contain all of the concentration series or replicate series in separate lists while the third contains all of the data; it should be noted that kinases in the third list are specified whether or not they are from a concentration or a replicate series.

```
conc_runs <- list(
  "AATK" = AATK,
  "ABL2" = ABL2_310,
  #"AKT1 First Run" = AKT1_414,
  #"AKT1 Second Run" = AKT1_510,
  "AKT1" = AKT1,
  "AKT2 First Run" = AKT2_221,
  "AKT2 Second Run" = AKT2_415,
  "AKT2 Third Run" = AKT2_509,
  "AKT3 First Run" = AKT3_416,
  "AKT3 Second Run" = AKT3_508,
```

```
rep_runs <- list(
  "ABL2" = ABL2_317,
  "BLK" = BLK_321,
  "CAMK1" = CAMK1_620,
  "CAMK2A" = CAMK2A_621,
  "CAMK2B" = CAMK2B_709,
  #"CAMK4" = CAMK4_710,
  "CSK" = CSK_511,
  "DDR2" = DDR2_723,
  "DMPK" = DMPK_505,
  "DYRK1B" = DYRK1B_509,
```

```
kinase_master_list <- list(
          "AATK Concentration" = AATK,
                           "ABL2 Concentration" = ABL2_310,
                           "ABL2 Replicate" = ABL2_317,
                           #"AKT1 Concentration 1" = AKT1_414,
                           #"AKT1 Concentration 2" = AKT1_510,
                           "AKT1 Concentration" = AKT1,
                           "AKT2 Concentration 1" = AKT2_221,
                           "AKT2 Concentration 2" = AKT2_415,
                           "AKT2 Concentration 3" = AKT2_509,
                           "AKT3 Concentration 1" = AKT3_416,
                           "AKT3 Concentration 2" = AKT3_508,
                           "AURKA Concentration" = AURKA,
                           "BLK Concentration" = BLK_311,
                           "BLK Replicate" = BLK_321,
```

More information on how these lists translate into interactive functionality can be found in section 3.

2-4 Adding New Kinase Datasets

To add new recombinant kinase data to Kinopedia, first find the 'run' folder within the main directory (section 2a). To add a new run, simply copy one of the subfolders within the 'run' directory and rename it. Once this is complete, rename the 'run_#_setup.R' script within the new folder and delete the *signal minus background* and *signal saturation files*, these will need to be replaced with the new KRSA files from the run you are adding. After adding the new files, the script is ready to edit.

The first section of each 'setup.R' script involves the use of the 'name_eval()' function; this function replaces the sample names with 'High, Medium, Low, and CTL' for concentration series or 'Rep1, Rep2, Rep3, and CTL' for replicates. Because not every set of data contains the same sample order, it is important to make sure that the 'vec' argument in the script file is correct. To check this, right click the *sample annotation* file that came with the kinase run and open it using Microsoft Excel.

After this, the rest of the script is relatively straightforward to edit. Following the template, the barcodes should be changed to match that of the run being added. The purpose of this section is to generate separate files for each of the kinases on the chip. After this, the names of the files being created should be changed to that of the kinase in question. Finally, change the values of the concentration vectors located towards the bottom of each section of the script. After doing all of this, copy the entire script and place it in the R terminal; it should automatically execute and generate the files.

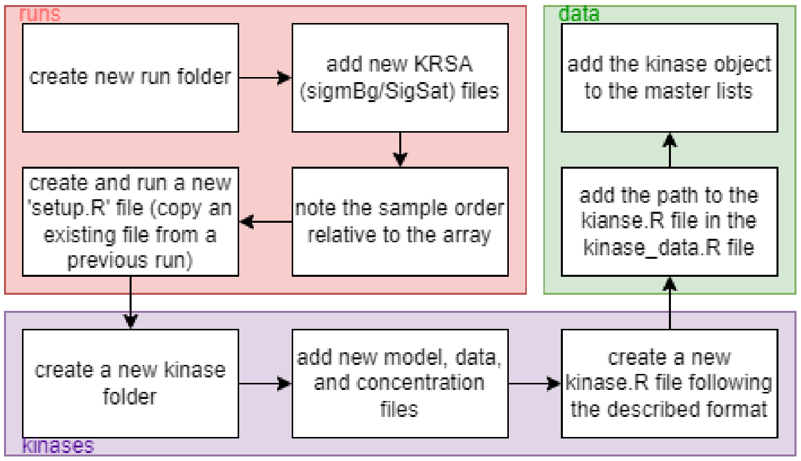
Adding the data to Kinopedia is largely the same as generating the new files. From the main directory, located the ‘kinases’ folder and open it. Inside are subfolders named after different kinases. If the kinase being added is not already present, add a new folder; it if is present, locate the existing kinase folder. From the ‘run’ folder in the last step, move the model, data, and concentration files into the new kinase folder. In the terminal, create a new script file and save it as *[kinase_name].R* to the kinase folder. Use the example shown in chapter 2b to fill out the script. If there are multiple sets of data for each kinase, the template should be copied and adjusted as necessary; all the datasets for each kinase should be located on the same script file.

After the new script has been created, head back to the main directory and locate the *data* folder. There are several scripts here, the most important being kinase_data.R. Open this file and add the path to the new kinase’s script file.

This adds the kinase object to the program, however, there are additional steps that must be taken. Several parts of Kinopedia rely on ‘master lists’, that are, lists that contain every dataset currently in the database. There are several of these master lists, and each one must be updated separately. In the case of recombinant kinases, there is another list that must be updated first; which list is updated depends on the run being uploaded- if the concentration of protein varies in each well (high, med, low), then the ‘conc_runs’ list is updated; if the concentration does not vary between wells, then the ‘rep_runs’ list is updated.

In either case, both types of data are added to the master list, with the type (concentration or replicate) being specified.

Once the kinase master list has been updated, the kinase is now registered in the database. A summary of this process is outlined in the following chart:


runs
create new run folder
add new KRSA (sigmBg/SigSat) files
note the sample order relative to the array
create and run a new 'setup.R' file (copy an existing file from a previous run)
data
add the kinase object to the master lists
add the path to the kianse.R file in the kinase_data.R file
kinases
create a new kinase folder
add new model, data, and concentration files
create a new kinase.R file following the described format

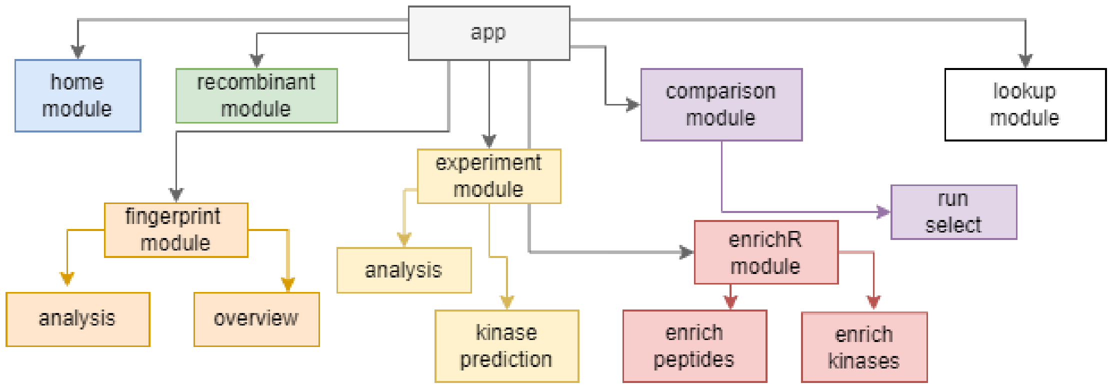
Although this tutorial was written with recombinant datasets in mind, a similar process can be conducted for both fingerprint and experimental datasets. These datasets require more manual modifications due to the non-standard naming and grouping conventions.

3. Rshiny Structure

The main program of Kinopedia is coded to work within the Rshiny framework. This is done through a system known as *reactive programming* which allows variables to be updated in real time. A detailed guide to Rshiny can be found at https://mastering-shiny.org/, however, this section will briefly go over how this system is integrated into Kinopedia.

3-1 Modules

Modules are the highest level of organization when it comes to Kinopedia. Modules encapsulate a majority of the functions such as viewing heatmaps, running correlation analyses, and conducting pathway analyses. Scripts for each module are contained in the 'modules' folder while the script that loads each module is found in the 'data' folder. Each module consists of a UI and a server component. This is the core of Rshiny, with the UI determining how the app appears and what the user can interact with and the server determining what outputs are generated. The diagram below shows how the hierarchy of modules is arranged in Kinopedia.



Each of the primary modules connects to app.R; additionally, several of these modules contains one or more sub-modules (i.e., the fingerprint module has both 'fingerprint_overview.R' and 'fingerprint_analysis.R' associated with it). The purpose of this structure is to reduce code complexity. For instance, the entirety of the fingerprint module is rather condensed:

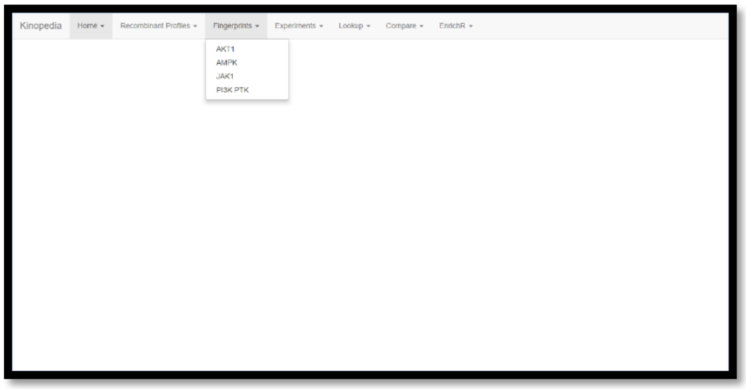
```
fingerUI <- function(id, runs, name){
  tabPanel(
    name,
    tabsetPanel(
      fingerOverviewUI(NS(id, "cell_over"), runs = runs),
      fingerAnalysisUI(NS(id, "cell_analyze"), runs = runs)
    )
  )
}

fingerServer <- function(id, runs, map){
  moduleServer(id, function(input, output, session){
    fingerOverviewServer("cell_over", runs = runs, map = map)
    fingerAnalysisServer("cell_analyze", runs, map = map)
  })
}
```

With the functional portion of the code being sequestered into sub-modules:

```
fingerOverviewServer <- function(id, runs, map){
  moduleServer(id, function(input, output, session){
    run <- reactiveValues(userInput = NULL)
    runChoice <- reactive({
      runs[[input$run_choice]]
    })
    observeEvent(runChoice(),{
      run$userInput <- runChoice()
    })
    signal <- reactive({
      input$sig_set
    })
```

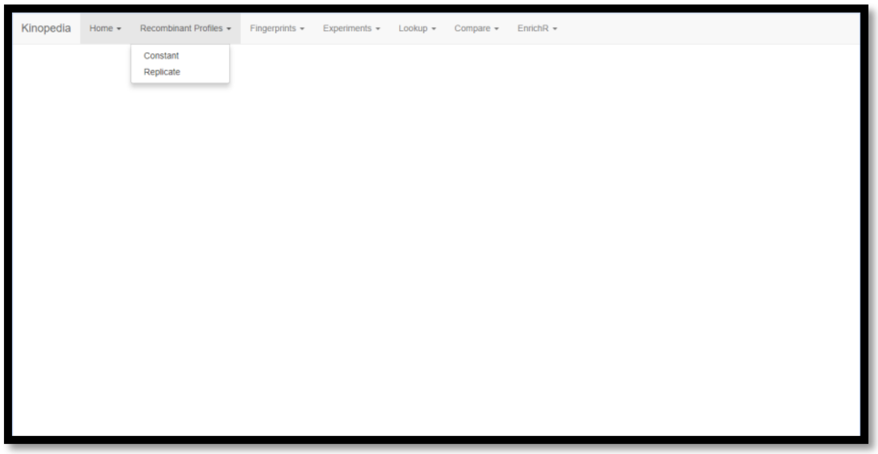
When we run the app, we can better understand the practical use of this complex setup. The primary modules appear from a top feature bar while each of the sub-modules are accessible from tabs contained within each primary module. This reflects the structure outlined in figure 3.

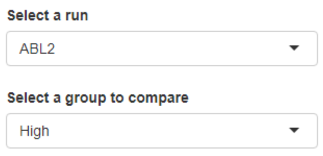
```
fingerOverviewServer <- function(id, runs, map){
  moduleServer(id, function(input, output, session){
    run <- reactiveValues(userInput = NULL)
    runChoice <- reactive({
      runs[[input$run_choice]]
    })
    observeEvent(runChoice(),{
      run$userInput <- runChoice()
    })
    signal <- reactive({
      input$sig_set
    })
```



3-2 Primer on Reactivity

In the process of adding new data, a step where the kinase object is added to one of two lists (replicate or concentration) was mentioned. The purpose of this is to allow the user to easily select a kinase from the drop-down menu.




Select a run
ABL2
Select a group to compare
High

Once selected, the kinase's details such as groups, concentration, and thresholds populate the rest of the interactive elements on the page. This is where the S4 classes come into play as all of these details can be packaged into a single object then passed to a list without creating a complex set of nested lists.

The code segment below is a section of the recombinant module UI. This specific segment is for the two drop-down menu boxes seen to the right. It should be noted that 'runs' refers to the list of kinases in the database (the replicate or concentration lists) from which the user can select a specific kinase.

```
selectInput(NS(id, "run_choice"),
            "Select a run",
            choices = names(runs),
            selected = "ABL2"),
selectInput(NS(id, "group_choice"),
            "Select a group to compare",
            choices = NULL,
            ),
```

On its own, the UI doesn't do anything. The server, shown below, is where the bulk of the functionality comes into play. Although this section of code appears rather complex, it is actually simple in execution.

```
run <- reactiveValues(userInput = NULL)
runChoice <- reactive({
  runs[[input$run_choice]]
})
observeEvent(runChoice(),{
  run$userInput <- runChoice()
  choices <- run$userInput@groups
  updateSelectInput(inputId = "group_choice",
                    choices = choices)
})
```

The first thing that is set up is a reactiveValues object. This object acts as a ‘container’ in which we can set various values; the execution of this is analogous to a list. Here, we are adding an element called ‘userInput’ to store the kinase selected by the user. The ‘reactive( )’ function creates a check that monitors any change to the specified input; in this case, the input is ‘run_choice’, which is the drop-down box containing kinases as seen in the UI. Upon detecting a change, reactive( ) crates a temporary reactive variable called runChoice( ). At first glance, this may seem redundant, as it would make more sense to directly place the selected kinase in the container. Unfortunately, reactive( ) is difficult to directly tie to reactiveValues. We can mitigate this through the use of observers that detect changes to reactive objects. When runChoice( ) is changed, an observer then places that kinase in the container which crates ‘run$userInput’, which is the kinase class object itself.

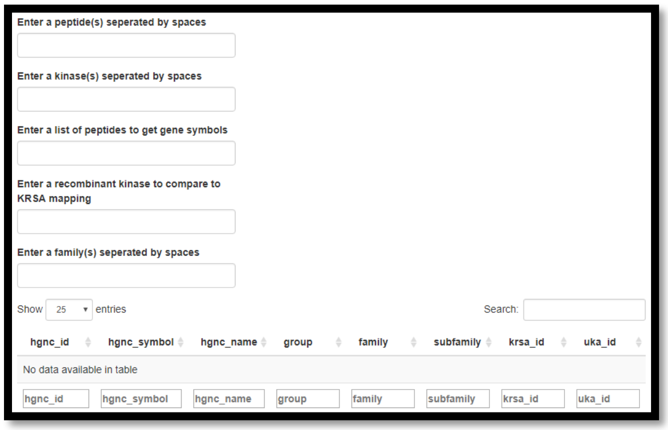
The final part of this code segment extracts the group names from the kinase object and sets those to the possible choices for the group select box. This pattern is repeated for most of the Kinopedia modules, therefore, understanding how this system works is paramount to understanding how most of the Kinopedia code works.

4. Specific Module Features

So far, we have explored both the database and the Rshiny components of Kinopedia, however, there is a third major component of Kinopedia- specific functionality of each module. This functionality includes figure generation, table generation, and any other components that are not controlled by Rshiny, S4 classes, or the main database.

4-1. The Lookup Module

The lookup module is used to browse information regarding the kinases in the database, however, it does not interact with the database itself. The initial view of this module is seen below:



Enter a peptide(s) seperated by spaces

Enter a kinase(s) seperated by spaces

Enter a list of peptides to get gene symbols

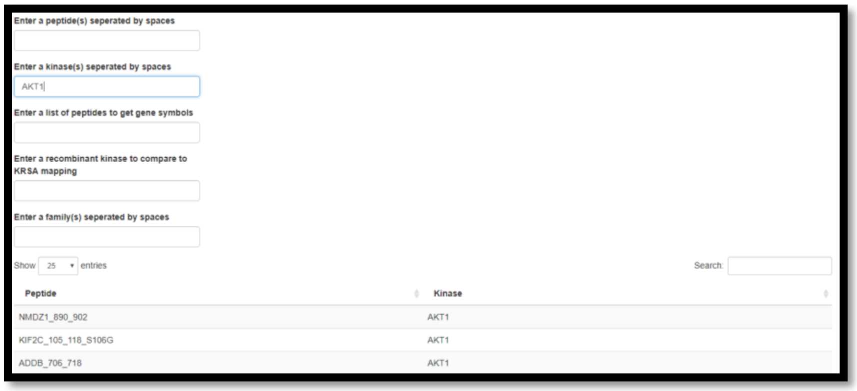
Enter a recombinant kinase to compare to KRSA mapping

Enter a family(s) seperated by spaces

Show 25 entries Search:

hgnc_id	hgnc_symbol	hgnc_name	group	family	subfamily	krsa_id	uka_id
No data available in table							
hgnc_id	hgnc_symbol	hgnc_name	group	family	subfamily	krsa_id	uka_id

When text is entered into one of the boxes, a table is returned. For instance, entering ‘AKT1’ returns a list of all peptides mapped to AKT1 based on the Kinopedia mapping file.



Enter a peptide(s) seperated by spaces

Enter a kinase(s) seperated by spaces

AKT1

Enter a list of peptides to get gene symbols

Enter a recombinant kinase to compare to KRSA mapping

Enter a family(s) seperated by spaces

Show 25 entries Search:

Peptide	Kinase
NMDZ1_890_902	AKT1
KIF2C_105_118_S106G	AKT1
ADDB_706_718	AKT1

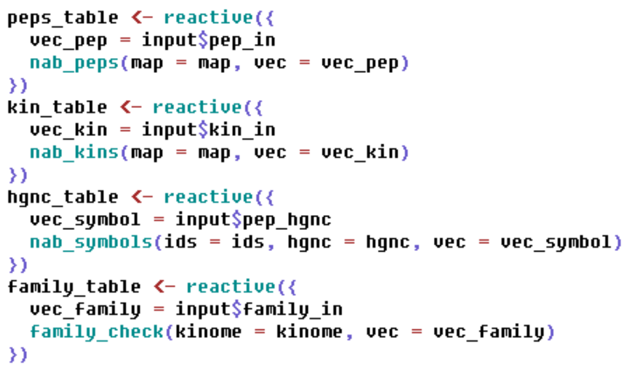
Previously, when the ‘data’ directory was explored in chapter 2, the .Rds files were ignored, however, they are integral to the lookup module. These files consist of tables that are read directly by Kinopedia through the ‘kinopedia_setup.R’ file. A breakdown of these files and the data that they contain is as follows:

- kinome_data: file from Creedenzymatic, contains KRSA-UKA connections, families, and HNGC symbols for the entire Kinome.

- Kinopedia_PTK_map: assigns peptides to recombinant PTKs based on the Kinopedia database.
- Kinopedia_STK_map: assigns peptides to recombinant STKs based on the Kinopedia database
- KRSA_ptk_map: assigns peptides to PTKs based on in-silico assignments; derived from the KRSA package.
- KRSA_stk_map: assigns peptides to STKs based on in-silico assignments; derived from the KRSA package.
- ptk_hgnc_map: connects PTK hgnc symbols to their corresponding UniProt IDs.
- ptk_id_map: connects PTK peptide names to their corresponding UniProt IDs.
- stk_hgnc_map: connects STK hgnc symbols to their corresponding UniProt IDs.
- stk_id_map: connects STK peptide names to their corresponding UniProt IDs.

Returning to the Kinopedia mapping files, we can get a better understanding as to why this component is independent from the database despite being a way to obtain database information. The reason for this is that the mapping files are manually curated; this means that peptide assignments are manually determined by the user (see chapter 5 more information on determining peptide assignments). Due to this, it is important to update the mapping files each time changes occur in the database. These files can be updated by loading them into the R console through the read.Rds( ) function and appending new data to them; once complete, the updated mapping file can be generated through the save.Rds( ) function.

Programmatically, the lookup module works off of a series of functions that subset an entry table (from the .Rds files) based on a set of user-entered words. Each of these subset tables is saved as a reactive object.



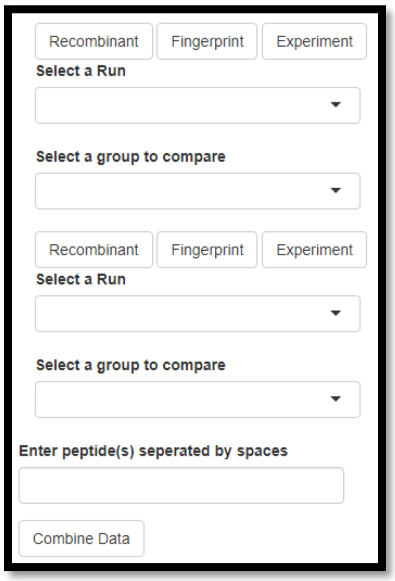
```
peps_table <- reactive({
  vec_pep = input$pep_in
  nab_peps(map = map, vec = vec_pep)
})
kin_table <- reactive({
  vec_kin = input$kin_in
  nab_kins(map = map, vec = vec_kin)
})
hgnc_table <- reactive({
  vec_symbol = input$pep_hgnc
  nab_symbols(ids = ids, hgnc = hgnc, vec = vec_symbol)
})
family_table <- reactive({
  vec_family = input$family_in
  family_check(kinome = kinome, vec = vec_family)
})
```

In order to make the module's UI efficient, a single table output is created. This output is tied to the container 'request$table'. When an input is detected in one of the boxes, the corresponding reactive table is placed into this container.

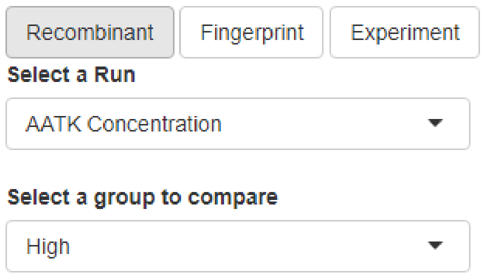
```
request <- reactiveValues(table = NULL)
observeEvent(input$pep_in,{
  request$table <- peps_table()
})
peps <- reactive({
  kin_table()$Peptide
})
observeEvent(input$kin_in,{
  request$table <- kin_table()
})
kins <- reactive({
  peps_table()$Kinase
})
```

4-2 The Comparison Module

The comparison module is used to directly compare different data sets. Comparisons are done through the LFC values from specific groups within the datasets. This is as rather complicated module with several different options.



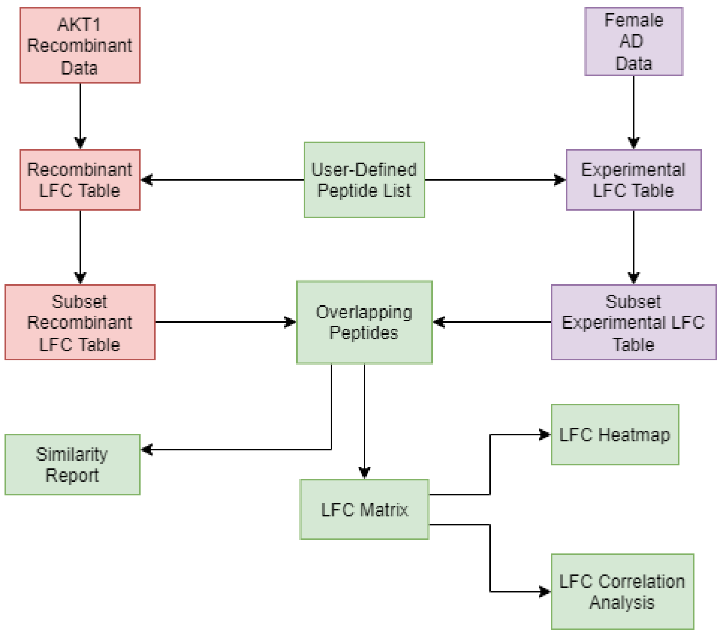
On the initial view, there are two identical selection fields. The buttons on the top are used to toggle between different types of data. Clicking on one of the buttons populates the other fields from which a run and group can be selected.



Select a Run: AATK Concentration

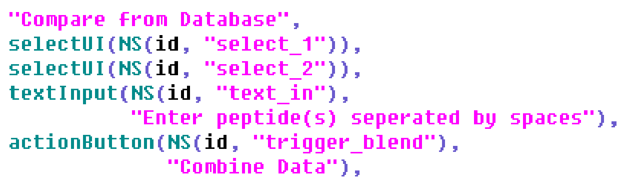
Select a group to compare: High

The text entry field towards the bottom is used to input the peptides whose LFC values will be used for the comparison. The following table can be used to better understand how this process works:

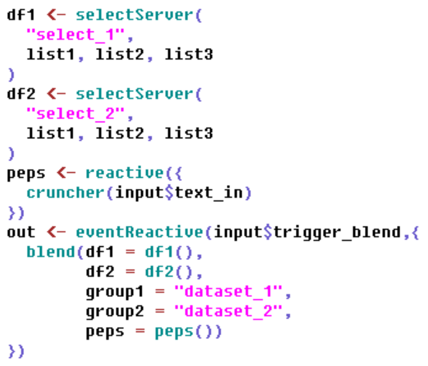


Essentially, an LFC table is generated for each of the selected datasets. Each of these tables is then subset using the list of user-entered peptides. Peptides from this subset that are found in both datasets are then used to construct a matrix from which further analyses can be conducted.

In order to better understand how this process works, we can explore the code. At the start of the code, we can see a sub-module distributed across both the UI and Server.



```
"Compare from Database",
selectUI(NS(id, "select_1")),
selectUI(NS(id, "select_2")),
textInput(NS(id, "text_in"),
          "Enter peptide(s) seperated by spaces"),
actionButton(NS(id, "trigger_blend"),
             "Combine Data"),
```



```
df1 <- selectServer(
  "select_1",
  list1, list2, list3
)
df2 <- selectServer(
  "select_2",
  list1, list2, list3
)
peps <- reactive({
  cruncher(input$text_in)
})
out <- eventReactive(input$trigger_blend,{
  blend(df1 = df1(),
        df2 = df2(),
        group1 = "dataset_1",
        group2 = "dataset_2",
        peps = peps())
})
```

The selection submodule is responsible for generating both of the LFC tables. This module works similarly to the other modules (recombinant, fingerprint, and experimental viewers). Of note is the presence of ‘list’ arguments in the function definition of the server:

```
selectServer <- function(id, list1, list2, list3){
  moduleServer(id, function(input, output, session){
    dataType <- reactiveValues(select = NULL)
    run <- reactiveValues(userInput = NULL)
    run <- reactiveValues(control = NULL)
    observeEvent(input$select_recomb,{
      dataType$select <- list1
      choices1 <- names(list1)
      updateSelectInput(inputId = "run_choice",
                        choices = choices1)

    })
```

These list arguments are the previously discussed 'master lists' that contain all of the data within the database. It should be noted that arguments utilized by sub-modules are also defined within the signature of the main function. These arguments are set in the 'app.R' file where the app as a whole is constructed:

```
compDatabaseServer("CE", list1 = kinase_master_list,
                         list2 = fingerprint_master_list,
                         list3 = experiment_master_list)
```

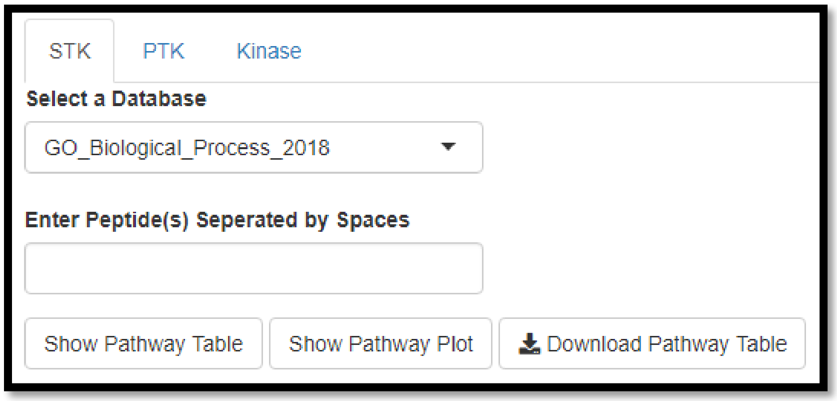
The remainder of the module consists of standard table/plot inputs and outputs. Descriptions of each of the functions within the rest of this module can be found in chapter 5.

4-3 The EnrichR Module

The EnrichR module is used to conduct pathway analysis; the actual pathway analysis is done through an R package written by Wajid Jawaid; documentation for this package can be found here: https://cran.r-project.org/web/packages/enrichR/vignettes/enrichR.html

The enrichR r package contains several databases that can be queried; currently, Kinopedia is set up to utilize eight unique databases. Additional databases can be added to Kinopedia by finding the 'dbs' object in the file 'kinopedia_setup.R'.

Functionally, the enrichR module operates in the same manner as the lookup module. The main page contains tabs for different categories of data, a dropdown menu for selecting a database, and several options to view a table, a plot, and downloading the table.



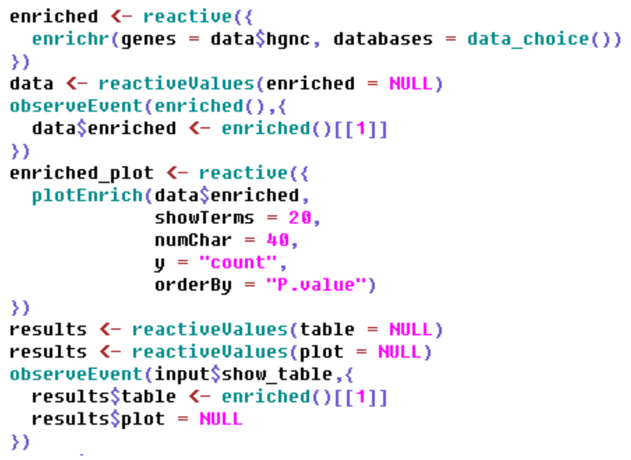
The actual code itself is relatively straightforward. Each of the three tabs is controlled by its own separate sub-module that loads each of the required arguments from the setup file.

```
enrichUI <- function(id){
  tabPanel(
    "EnrichR",
    tabsetPanel(
      enrichPepUI(NS(id, "enrich_stk"), name = "STK", dbs = dbs),
      enrichPepUI(NS(id, "enrich_ptk"), name = "PTK", dbs = dbs),
      enrichKinUI(NS(id, "enrich_kinase"), dbs = dbs)
    )
  )
}

enrichServer <- function(id, id_stk, id_ptk, hgnc_stk, hgnc_ptk, dbs){
  moduleServer(id, function(input, output, session){
    enrichPepServer("enrich_stk",
                    pep_map = id_stk,
                    hgnc_map = hgnc_stk,
                    dbs = dbs)
    enrichPepServer("enrich_ptk",
                    pep_map = id_ptk,
                    hgnc_map = hgnc_ptk,
                    dbs = dbs)
    enrichKinServer("enrich_kinase",
                    dbs = dbs)
  })
}
```

Looking at the ‘pep_enrich.R’ file, it can be seen that the sub-modules are a bit more complex, however, it can be easily broken down. Most of the UI consists entirely of components related to displaying the enrichR results with a text-input and database selector field.

```
enrichPepUI <- function(id, name, dbs){
  tabPanel(
    name,
    selectInput(NS(id, "database_select"),
                "Select a Database",
                choices = dbs,
                selected = dbs[3]
    ),
    textInput(NS(id, "text_in"),
              "Enter Peptide(s) Seperated by Spaces"),
    actionButton(NS(id, "show_table"),
                 "Show Pathway Table"),
    actionButton(NS(id, "show_plot"),
                 "Show Pathway Plot"),
    downloadButton(NS(id, "download"),
                   "Download Pathway Table"),
    dataTableOutput(NS(id, "enrich1")),
    plotOutput(NS(id, "enrich2"))
  )
}
```

Although the server appears to be complicated on the surface, a majority of it follows previously established patterns of reactivity. Enrichr itself only accepts HNGC symbols; the portion of the code that assigns these symbols to peptides works exactly the same as it does for the lookup module. The key difference is the inclusion of the enrichrR portion; most of this is controlled by the *enrichr* function itself which comes from the enrichr package. The ‘plotEnrich( ) function converts the initial table results to a plot; both the table and plot forms of the results are stored in the ‘results’ container.



```
enriched <- reactive({
  enrichr(genes = data$hgnc, databases = data_choice())
})
data <- reactiveValues(enriched = NULL)
observeEvent(enriched(),{
  data$enriched <- enriched()[[1]]
})
enriched_plot <- reactive({
  plotEnrich(data$enriched,
             showTerms = 20,
             numChar = 40,
             y = "count",
             orderBy = "P.value")
})
results <- reactiveValues(table = NULL)
results <- reactiveValues(plot = NULL)
observeEvent(input$show_table,{
  results$table <- enriched()[[1]]
  results$plot = NULL
})
```

Something to note is the inclusion of a download function; this function is to download the table to a .csv file, however, there are multiple issues with this function that need to be resolved in the future. First, the filetype must be specified by manually adding '.csv' to the filename. Second, the file will only download when the table is being displayed; the reason for this is that the download handler is drawing from the 'data' container. If the plot is being displayed then the container is filled with the plot which cannot be downloaded.

5. Function Guide

The remainder of this handbook will be dedicated to explaining the many functions that can be found within the *functions* directory of the app; the main purpose of including this section is to provide better clarity for future package development. In this section, there are a few function types:

- Structural: used for rendering Rshiny UI elements
- Wrapper: used to several different functions into a single function; normally outputs a list object of several outputs
- Constructor: builds a new S4 object of a defined class
- Utility: automates a simple task such as counting items in a list
- Complex: functions that either perform complicated operations or contain many different components
- Other: functions that do not fit the other four categories
- Generic: functions that serve as generics for S4 classes

5-1 Normal Functions

blend:

- Generates a combined table of two independent LFC tables, a peptide distribution report, and a heatmap of LFC values.
- Type: complex
- Inputs: df1 (first LFC table), df2 (second LFC table), group1 (name of first group), group2 (name of second group), peps (list of peptides of interest)
- Outputs: out (list object containing the combined dataframe, the distribution dataframe, and a heatmap)
- Children: blender_heatmap, blender_matrix

blender_heatmap:

- Creates a heatmap using a matrix of LFC values
- Type: other
- Inputs: data (matrix object), … (additional arguments compatible with R heatmaps)
- Outputs: heatmap
- Parent: blend

blender_matrix:

- Creates a matrix of LFC values
- Type: other
- Inputs: data (combined dataframe of the two LFC tables of interest), peptides (list of peptides of interest)
- Outputs: a matrix of LFC values

- Parent: blend

build_kinase_conc:

- Creates a new kinase object for a concentration series
- Type: constructor
- Inputs: data (data.Rds file), model (model.Rds file), concs (concs.Rds file)
- Outputs: a kinase object

build_kinase_rep:

- Creates a new kinase object for a replicate series
- Type: constructor
- Inputs: data (data.Rds file), model (model.Rds file), concs (concs.Rds file)
- Outputs: a kinase object

cor_pearson[1]:

- Runs a correlation analysis between two LFC dataframes
- Type: other
- Inputs: y1 (first group), y2 (second group), ylab1 (name of first group), ylab2 (name of second group), method (type of test to use)
- Outputs: a list containing two quality reports and a complex results object used in further functions
- Parent:

cor_plot_maker:

- Generates a correlation scatterplot and two quality report scatterplots
- Type: wrapper
- Inputs: df (combined LFC dataframe), method (type of test to use)
- Outputs: a list containing the scatterplot and two quality plots in the form of ggplot objects
- Children: cor_scatter

cor_plot_rodeo:

- Set of radio buttons used to toggle between plot displays, specifically, those produced by cor_plot_maker
- Type: structural
- Inputs: id (id used in Rshiny)
- Outputs: N/A

cor_scatter:

- Generates a scatterplot of a correlation between two LFC dataframes
- Type: other

- Inputs: df (combined dataframe), x (data column for the x-axis), y (data column for the y-axis), xlab (label of the x-axis), ylab (label for the y-axis), method (type of correlation test)
- Outputs: plot (ggplot object)
- Parent: cor_plot_maker

cov_preprocess[2]:

- Counts the number of kinases that map to a set of given peptides
- Type: complex
- Inputs: map (mapping table to use), peps (list of peptides of interest)
- Outputs: perc_df (dataframe of the counts)
- Parent:
- Children: get_counts, generate_perc_df

cruncher:

- Splits a single string into a vector of multiple strings based on spaces (“ “)
- Type: utility
- Inputs: string (single string)
- Outputs: string (vector of strings)
- Parents: family_check,

diff_df_extend:

- Produces a set of data transformed tables from a given table
- Type: wrapper
- Inputs: df (a dataframe of LFC values)
- Outputs: out (a list containing the original dataframe, a z-normalized version, and a log transformed version)
- Children: zed_transformer, log_transformer

export_data[3]:

- Filters data and creates a table of LFC values
- Type: complex
- Inputs: groups (groups to compare), data (data object derived from data.Rds), model (linear model derived from model.Rds), sig_thresh (signal cutoff), rseq_thresh (r-square cutoff)
- Outputs: diff_df_fil (filtered LFC table)
- Children: mega_filter_grouped, get_LFC

family_check:

- Looks for what type of family a particular kinase comes from
- Type: utility
- Inputs: kinome (kinome file from Creedenzymatic, vec (user-derived list of kinases)
- Outputs: df (report of kinase families)

- Children: cruncher

generate_perc_df^2:

- Generates a table of percent coverage for a set of kinases
- Type: other
- Inputs: kinases (list of kinases of interest), counts_sub (number of peptides of interest mapped to those kinases), counts_all (number of all peptides mapped to those kinases)
- Outputs: df (percent coverage report)
- Parent: cov_preprocess

generate_plot_df^2:

- Generates the dataframe used to construct the coverage plot
- Type: other
- Inputs: kinases (list of kinases of interest), data (table containing the percent coverage report)
- Outputs: data (table that contains information relating to subset counts and total counts; used to construct a stacked barplot)
- Children: generate_sub_df, generate_total_df
- Parents: swoosh_plot

generate_sub_df^2:

- Generates the top part of the plot dataframe for the coverage section containing the peptides of interest
- Type: other
- Inputs: kinases (list of kinases of interest), percent (percent of the subset covered), ratio (ratio of the subset to the total)
- Outputs: df2 (subset coverage dataframe)
- Parent: generate_plot_df

generate_total_df^2:

- Generates the bottom part of the plot dataframe for the coverage section containing the total number of peptides each kinase is mapped to
- Type: other
- Inputs: kinases (list of kinases of interest), percent (percent of the total not covered by the subset)
- Outputs: df (total coverage dataframe)
- Parent: generate_plot_df

get_counts:

- Counts the number of peptides mapped to each kinase in a list of kinases
- Type: utility
- Inputs: df (dataframe containing the kinases of interest and their associated peptides), kinases (list of kinases of interest)

- Outputs: counts (vector consisting of peptide count information)
- Parent:

get_ctl_peps:

- Obtains the list of peptides in the control group of a kinase data object
- Type: utility
- Inputs: data (dataframe)
- Outputs: p2 (vector of control group peptides)

get_lfc:

- Generates the LFC table for a given comparison
- Type: other
- Inputs: data (dataframe of slope values), groups (groups to compare), peps (peptides of interest)
- Outputs: an LFC table
- Parent: export_data

global_annotation:

- Extracts the sample names from a dataframe
- Type: utility
- Inputs: data (dataframe to obtain sample names from)
- Outputs: a dataframe with the sample names set to the rownames

global_heatmap:

- Generates a heatmap of the global data
- Type: other
- Inputs: data (dataframe containing slope values), peptides (list of peptides of interest), … (optional arguments for heatmaply)
- Outputs: a global heatmap
- Children: global_matrix, global_annotation
- Parent: global_heatmap_maker

global_heatmap_grouped:

- Generates a grouped heatmap of the global data
- Type: other
- Inputs: data (dataframe containing slope values), peptides (list of peptides of interest), … (optional arguments for heatmaply)
- Outputs: a grouped global heatmap
- Children: global_matrix_grouped
- Parent: global_heatmap_maker

global_heatmap_maker[3]:

- Creates all of the global heatmaps and performs a filtering process

- Type: wrapper
- Inputs: model (model .Rds file), data (data .Rds file), sig_thresh (the signal threshold), rseq_thresh (the rsquare threshold)
- Outputs: heatmaps (a list of heatmaps and a NULL default)
- Children: mega_filter_global, global_heatmap, global_heatmap_grouped

global_matrix:

- Generates a matrix of slope values for global heatmaps
- Type: other
- Inputs: data (dataframe of slope values), peptides (list of peptides of interest)
- Outputs: a matrix of slope values
- Parent: global_heatmap

global_matrix_grouped:

- Generates a matrix of slope values for global heatmaps with group assignments
- Type: other
- Inputs: data (dataframe of slope values), peptides (list of peptides of interest)
- Outputs: a matrix of slope values with group assignments
- Parent: global_heatmap_grouped

group_annotation:

- Extracts the sample names from a dataframe
- Type: utility
- Inputs: data (dataframe to obtain sample names from)
- Outputs: a dataframe with the sample names set to the rownames

group_heatmap:

- Generates a heatmap of the global data
- Type: other
- Inputs: data (dataframe containing slope values), groups (list of groups to compare) peptides (list of peptides of interest), … (optional arguments for heatmaply)
- Outputs: a heatmap for specific groups
- Children: group_matrix, global_annotation
- Parent: group_heatmap_maker

group_heatmap_maker:

- Creates a heatmap of slope values for specified group
- Type: wrapper
- Inputs: model (model .Rds file), groups (groups to include), diff_df (data containing the slope values)
- Outputs: heatmaps a group heatmap
- Children: group_heatmap

group_matrix:

- Generates a matrix of slope values for group heatmaps
- Type: other
- Inputs: data (dataframe of slope values), peptides (list of peptides of interest)
- Outputs: a matrix of slope values
- Parent: group_heatmap

heat_builder[3]:

- Used for the 'make_heatmaps' generic; builds both global and group heatmaps
- Type: complex
- Inputs: model (model .Rds file), data (data .Rds file), sig_thresh (signal cutoff threshold), rseq_thresh (rsqure cutoff threshold), groups (groups to compare for group comparisons), diff_df (LFC table containing the peptides)
- Outputs: heatmaps (list of global, global grouped, and group heatmaps)
- Children: mega_filter_global, global_heatmap, global_heatmap_grouped, group_heatmap_maker
- Parent: make_heatmaps (generic)

heatmap_rodeo:

- UI element that creates the buttons used to toggle between heatmaps
- Type: structural
- Inputs: id (Rshiny ID)
- Outputs: N/A

heatmap_rodeo2:

- UI element that creates the buttons used to toggle between heatmaps; unlike the first heatmap_rodeo function, this one lacks the option for group heatmaps
- Type: structural
- Inputs: id (Rshiny ID)
- Outputs: N/A

kin_sort:

- Finds common kinases between KRSA and UKA; used in conjunction with the kinome dataframe from Creedenzymatic
- Type: utility
- Inputs: df (kinome dataframe)
- Outputs: kins (list of kinases common to both KRSA and UKA)

kinase_juicer:

- Creates both a quartile figure and combined heatmap from KRSA and UKA files; part of the Creedenzymatic workflow
- Type: complex

- Inputs: krsa_df (KRSA table), uka_df (UKA table)
- Outputs: out (list containing a combined Creedenyatic table and a quartile figure)

krsa_compare:

- Determines the ratio of mapped to a set of kinases between KRSA and kinopedia mapping.
- Type: complex
- Inputs: kinase (kinases of interest), krsa_map (KRSA mapping file), kinopedia_map (kinopedia mapping file)
- Outputs: out
- Children: sorter

lfc_rodeo:

- Set of buttons used to toggle between various LFC tables
- Type: structural
- Inputs: id (Shiny module ID)
- Outputs: N/A

log_transformer:

- Applies a log transformation to a two-column matrix of values
- Type: utility
- Inputs: df (matrix containing two groups)
- Outputs: df (log-transformed version of the input dataframe)
- Parent: diff_df_extend

low_filter_global:

- Filters out peptides that do not meet a specific signal value; currently set to remove the control group before filtering
- Type: utility
- Inputs: data (dataframe with the slope values), sig_thresh (value to use for the signal threshold)
- Outputs: peps (list of filtered peptides from non-control groups combined with all peptides from the control group)
- Children: get_ctl_peps
- Parents: mega_filter_global

low_filter_grouped:

- Filters out peptides that do not meet a specific signal value; only applied to specified groups
- Type: utility
- Inputs: data (dataframe with the slope values), sig_thresh (value to use for the signal threshold), groups (groups of interest)
- Outputs: p (list of peptides from the specified groups that met the threshold)

- Parents: mega_filter_grouped

make_cov_plot[2]:

- Generates a coverage plot for a given input
- Type: other
- Inputs: data (dataframe containing the kinases/peptides as well as their respective ratios)
- Outputs: plot (a ggplot object)
- Parents: swoosh_plot

mega_filter_global[3]:

- Applies a set of filters to a dataset
- Type: wrapper
- Inputs: data (data .Rds file), model (scaled model), sig_thresh (signal cutoff value), rseq_thresh (slope linearity (rsquare) cutoff value)
- Outputs: p3 (filtered peptide list)
- Children: low_filter_global, nonlinear_filter_global, ref_filter
- Parents: heat_builder

nab_kins:

- Filters a given map for the list of user-inputted kinases
- Type: utility
- Inputs: map (mapping file to use), vec (vector of user submitted kinases)
- Outputs: map (filtered mapping file)
- Children: cruncher

nab_peps:

- Filters a given map for the list of user-inputted peptides
- Type: utility
- Inputs: map (mapping file to use), vec (vector of user submitted peptides)
- Outputs: map (filtered mapping file)
- Children: cruncher

nab_symbols[4,5]:

- Obtains the HGNC symbols for a user submitted list of peptides by first matching them to their UniProt IDs, then matching the IDs to the symbols
- Type: utility
- Inputs: ids (map containing IDs), hgnc (map containing HGNC symbols), vec (vector of user submitted peptides)
- Outputs: df (dataframe containing the peptides mapped to their corresponding HGNC symbols)
- Children: cruncher

name_eval:

- Standardizes the group names for a given kinase run; used in the pre-processing steps when generating the files for each kinase (model, data, concentrations, etc.)
- Type: utility
- Inputs: data (data loaded using the KRSA_read( ) function), vec (names to replace existing group names)
- Outputs: df (dataframe with replaced group names)

nonlinear_filter_global[3]:

- Filters out peptides that do not meet a given linearity (rsquare value threshold)
- Type: utility
- Inputs: data (linear model), rseq_thresh (rsquare threshold)
- Outputs: p (filtered list of peptides)
- Parents: mega_filter_global

nonlinear_filter_grouped[3]:

- Filters out peptides for a set of specified groups that do not meet a given linearity (rsquare value threshold)
- Type: utility
- Inputs: data (linear_model), rseq_thresh (rsquare threshold), groups (groups of interest)
- Outputs: p (filtered list of peptides)
- Parents: mega_filter_grouped

process_major_output[6]:

- Generates a new output for the ‘major_output’ class
- Type: constructor
- Inputs: df_recomb (recombinant dataframe), df_finger (fingerprint dataframe)
- Outputs: out (object of the major_output class)

ref_filter:

- Filters out reference peptides
- Type: utility
- Inputs: peptides (list of peptides produced in the previous two filtration steps)
- Outputs: new_pep (list of peptides with references removed)
- Parents: mega_filter_grouped, mega_filter_global

remove_ctl:

- Removes the control group from a dataframe
- Type: utility
- Inputs: data (dataframe)
- Outputs: df (dataframe without the control group)
- Parents: mega_filter_global

rsquare_slider:

- Allows the user to adjust the Rsquare threshold
- Type: structural
- Inputs: id (Shiny module ID)
- Outputs: N/A

shapiro_tester:

- Performs the Shaprio-Wilks test for normality on a given matrix
- Type: utility
- Inputs: df (matrix containing a set of values and the two groups to be compared)
- Outputs: out (table with the results including pvalues, means, and standard deviations)

signal_slider:

- Allows the user to adjust the signal threshold
- Type: structural
- Inputs: id (Shiny module ID)
- Outputs: N/A

sorter:

- Extracts peptides found only in specific groups
- Type: utility
- Inputs: total_peps (all peptides of interest), peps1 (all peptides of interest), peps2 (peptides found in second group), group1 (name of first group), group2 (name of second group), overlap (peptides found in both groups)
- Outputs: out (dataframe produced by the tabulator function)
- Children: tabulator
- Parent: krsa_compare

swoosh_plot[2]:

- Wrapper function to generate the coverage dataframe and then build the plot
- Type: wrapper
- Inputs: kinases (kinases to find the coverage of), data (dataframe containing both kinases and peptides)
- Outputs: plot (ggplot object)
- Children: generate_plot_df, make_cov_plot

tabulator:

- Calculates the ratio of peptide distribution from a set of given inputs
- Type: utility
- Inputs: total_peps (all peptides of interest), df1_only (only peptides found in first group), df2_only (only peptides found in second group), overlap_peps (peptides found in both groups), group1 (name of first group), group2 (name of second group)
- Outptus: df (dataframe showing the peptide distribution)
- Parent: sorter

threshold_slider:

- Allows the user to change the LFC threshold
- Type: structural
- Inputs: id (Shiny module ID)
- Outputs: N/A

zed_transformer:

- Applies a Z score transformation to a two-column matrix of values
- Type: utility
- Inputs: df (matrix containing two groups)
- Outputs: df (Z-transformed version of the input dataframe)
- Parent: diff_df_extend

Notes:

1. Should be renamed; Pearson correlation analysis is no longer the only type of correlation analysis that the program can perform.

2. May be removed in the future; the coverage plot generated by these functions is difficult to interpret conceptually

3. The ‘rseq’ argument in these functions should be renamed to ‘R2’ or ‘rsquare’ to avoid confusion with RNA sequencing

4. May be changed in a future update if a method of directly matching peptide names to their genes can be found; this would remove the intermediary step of first matching to UniProt IDs and simplify the code.

5. It should be noted that not all IDs have assigned symbols or were removed from the database; this means that not all submitted peptides may be reflected in the HGNC list.

6. This is fully depreciated; the ‘major_output’ class no longer exists.

5-2 Generics

These functions are part of the S4 object system and can be applied to multiple different object classes. Oftentimes, a generic will replace a function within the code making it important to understand what functions they are linked to.

global_heat:

- Associated functions: global_heatmap_maker
- Applicable classes: kinase, fingerprint

group_heat:

- Associated functions: group_heatmap_maker
- Applicable classes: kinase, experiment, fingerprint

fuse:

- Associated functions: kinase_juicer
- Applicable classes: experiment

lfc_table:

- Associated functions: export_data
- Applicable classes: kinase, experiment, fingerprint

make_heatmaps:

- Associated functions: heat_builder
- Applicable classes: kinase, fingerprint