If you build statistical or machine learning models, the recipes package can be useful for data preparation. A recipe object is a container that holds all the steps that should be performed to go from the raw data set to the set that is fed into model a algorithm. Once your recipe is ready it can be executed on a data set at once, to perform all the steps. Not only on the train set on which the recipe was created, but also on new data, such as test sets and data that should be scored by the model. This assures that new data gets the exact same preparation as the train set, and thus can be validly scored by the learned model.I will not dwell on how to use the package. Rather, I’d like to share what in my opinion is a good way to create new steps and checks to the package[1](https://edwinth.github.io/blog/recipes_blog/#fn:1). Use of the package is probably intuitive. Developing new steps and checks, however, does require a little more understanding of the package inner workings. With this procedure, or recipe if you like, I hope you will find adding (and maybe even contributing) your own steps and checks becomes easier and more organized.

Recipes Introduction

**An Example**

The packages contains a data set used to predict whether a person will pay back a bank loan. It has 13 predictor columns and a factor variable Status (the outcome). We will first separate the data into a training and test set:

library(recipes)

library(rsample)

library(modeldata)

data("credit\_data")

set.seed(55)

train\_test\_split <- initial\_split(credit\_data)

credit\_train <- training(train\_test\_split)

credit\_test <- testing(train\_test\_split)

Note that there are some missing values in these data:

vapply(credit\_train, **function**(x) mean(!is.na(x)), numeric(1))

*#> Status Seniority Home Time Age Marital Records Job*

*#> 1.000 1.000 0.998 1.000 1.000 1.000 1.000 0.999*

*#> Expenses Income Assets Debt Amount Price*

*#> 1.000 0.910 0.989 0.996 1.000 1.000*

Rather than remove these, their values will be imputed.

The idea is that the preprocessing operations will all be created using the training set and then these steps will be applied to both the training and test set.

**An Initial Recipe**

First, we will create a recipe object from the original data and then specify the processing steps.

Recipes can be created manually by sequentially adding roles to variables in a data set.

If the analysis only requires **outcomes** and **predictors**, the easiest way to create the initial recipe is to use the standard formula method:

rec\_obj <- recipe(Status ~ ., data = credit\_train)

rec\_obj

*#> Recipe*

*#>*

*#> Inputs:*

*#>*

*#> role #variables*

*#> outcome 1*

*#> predictor 13*

The data contained in the data argument need not be the training set; this data is only used to catalog the names of the variables and their types (e.g. numeric, etc.).

(Note that the formula method is used here to declare the variables, their roles and nothing else. If you use inline functions (e.g. log) it will complain. These types of operations can be added later.)

**Preprocessing Steps**

From here, preprocessing steps for some step *X* can be added sequentially in one of two ways:

rec\_obj <- step\_{X}(rec\_obj, arguments) *## or*

rec\_obj <- rec\_obj %>% step\_{X}(arguments)

step\_dummy and the other functions will always return updated recipes.

One other important facet of the code is the method for specifying which variables should be used in different steps. The manual page ?selections has more details but [dplyr](https://cran.r-project.org/package=dplyr)-like selector functions can be used:

* use basic variable names (e.g. x1, x2),
* [dplyr](https://cran.r-project.org/package=dplyr) functions for selecting variables: contains(), ends\_with(), everything(), matches(), num\_range(), and starts\_with(),
* functions that subset on the role of the variables that have been specified so far: all\_outcomes(), all\_predictors(), has\_role(),
* similar functions for the type of data: all\_nominal(), all\_numeric(), and has\_type(), or
* compound selectors such as all\_nominal\_predictors() or all\_numeric\_predictors().

Note that the methods listed above are the only ones that can be used to select variables inside the steps. Also, minus signs can be used to deselect variables.

For our data, we can add an operation to impute the predictors. There are many ways to do this and recipes includes a few steps for this purpose:

grep("impute$", ls("package:recipes"), value = TRUE)

*#> [1] "step\_bagimpute" "step\_knnimpute" "step\_lowerimpute"*

*#> [4] "step\_meanimpute" "step\_medianimpute" "step\_modeimpute"*

*#> [7] "step\_rollimpute"*

Here, *K*-nearest neighbor imputation will be used. This works for both numeric and non-numeric predictors and defaults *K* to five To do this, it selects all predictors and then removes those that are numeric:

imputed <- rec\_obj %>%

step\_knnimpute(all\_predictors())

*#> Warning: `step\_knnimpute()` was deprecated in recipes 0.1.16.*

*#> Please use `step\_impute\_knn()` instead.*

imputed

*#> Recipe*

*#>*

*#> Inputs:*

*#>*

*#> role #variables*

*#> outcome 1*

*#> predictor 13*

*#>*

*#> Operations:*

*#>*

*#> K-nearest neighbor imputation for all\_predictors()*

It is important to realize that the *specific* variables have not been declared yet (as shown when the recipe is printed above). In some preprocessing steps, variables will be added or removed from the current list of possible variables.

Since some predictors are categorical in nature (i.e. nominal), it would make sense to convert these factor predictors into numeric dummy variables (aka indicator variables) using step\_dummy(). To do this, the step selects all non-numeric predictors:

ind\_vars <- imputed %>%

step\_dummy(all\_nominal\_predictors())

ind\_vars

*#> Recipe*

*#>*

*#> Inputs:*

*#>*

*#> role #variables*

*#> outcome 1*

*#> predictor 13*

*#>*

*#> Operations:*

*#>*

*#> K-nearest neighbor imputation for all\_predictors()*

*#> Dummy variables from all\_nominal\_predictors()*

At this point in the recipe, all of the predictor should be encoded as numeric, we can further add more steps to center and scale them:

standardized <- ind\_vars %>%

step\_center(all\_numeric\_predictors()) %>%

step\_scale(all\_numeric\_predictors())

standardized

*#> Recipe*

*#>*

*#> Inputs:*

*#>*

*#> role #variables*

*#> outcome 1*

*#> predictor 13*

*#>*

*#> Operations:*

*#>*

*#> K-nearest neighbor imputation for all\_predictors()*

*#> Dummy variables from all\_nominal\_predictors()*

*#> Centering for all\_numeric\_predictors()*

*#> Scaling for all\_numeric\_predictors()*

If these are the only preprocessing steps for the predictors, we can now estimate the means and standard deviations from the training set. The prep function is used with a recipe and a data set:

trained\_rec <- prep(standardized, training = credit\_train)

trained\_rec

*#> Recipe*

*#>*

*#> Inputs:*

*#>*

*#> role #variables*

*#> outcome 1*

*#> predictor 13*

*#>*

*#> Training data contained 3340 data points and 322 incomplete rows.*

*#>*

*#> Operations:*

*#>*

*#> K-nearest neighbor imputation for Home, Time, Age, Marital, Records, Job, Expens... [trained]*

*#> Dummy variables from Home, Marital, Records, Job [trained]*

*#> Centering for Seniority, Time, Age, Expenses, Income, Assets,... [trained]*

*#> Scaling for Seniority, Time, Age, Expenses, Income, Assets,... [trained]*

Note that the real variables are listed (e.g. Home etc.) instead of the selectors (all\_numeric\_predictors()).

Now that the statistics have been estimated, the preprocessing can be *applied* to the training and test set:

train\_data <- bake(trained\_rec, new\_data = credit\_train)

test\_data <- bake(trained\_rec, new\_data = credit\_test)

bake returns a tibble that, by default, includes all of the variables:

class(test\_data)

*#> [1] "tbl\_df" "tbl" "data.frame"*

test\_data

*#> # A tibble: 1,114 × 23*

*#> Seniority Time Age Expenses Income Assets Debt Amount Price*

*#> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>*

*#> 1 1.09 0.924 1.88 -0.385 -0.131 -0.488 -0.295 -0.0817 0.297*

*#> 2 -0.977 0.924 -0.459 1.77 -0.437 0.845 -0.295 0.333 0.760*

*#> 3 -0.977 0.103 0.349 1.77 -0.783 -0.488 -0.295 0.333 0.00254*

*#> 4 -0.247 0.103 -0.280 0.231 -0.207 -0.133 -0.295 0.229 0.171*

*#> 5 -0.125 -0.718 -0.729 0.231 -0.258 -0.222 -0.295 -0.807 -0.854*

*#> 6 -0.855 0.924 -0.549 -1.05 -0.0539 -0.488 -0.295 0.436 -0.331*

*#> 7 2.31 0.924 0.349 0.949 -0.0155 -0.488 -0.295 -0.185 0.0475*

*#> 8 0.848 -0.718 0.529 1.00 1.40 -0.133 -0.295 1.58 1.69*

*#> 9 -0.977 -0.718 -1.27 -0.538 -0.246 -0.266 -0.295 -1.32 -1.65*

*#> 10 -0.855 0.514 -0.100 0.744 -0.540 -0.488 -0.295 -0.185 -0.800*

*#> # … with 1,104 more rows, and 14 more variables: Status <fct>, Home\_X1 <dbl>,*

*#> # Home\_X2 <dbl>, Home\_X3 <dbl>, Home\_X4 <dbl>, Home\_X5 <dbl>,*

*#> # Marital\_X1 <dbl>, Marital\_X2 <dbl>, Marital\_X3 <dbl>, Marital\_X4 <dbl>,*

*#> # Records\_X1 <dbl>, Job\_X1 <dbl>, Job\_X2 <dbl>, Job\_X3 <dbl>*

vapply(test\_data, **function**(x) mean(!is.na(x)), numeric(1))

*#> Seniority Time Age Expenses Income Assets Debt*

*#> 1 1 1 1 1 1 1*

*#> Amount Price Status Home\_X1 Home\_X2 Home\_X3 Home\_X4*

*#> 1 1 1 1 1 1 1*

*#> Home\_X5 Marital\_X1 Marital\_X2 Marital\_X3 Marital\_X4 Records\_X1 Job\_X1*

*#> 1 1 1 1 1 1 1*

*#> Job\_X2 Job\_X3*

*#> 1 1*

Selectors can also be used. For example, if only the predictors are needed, you can use bake(object, new\_data, all\_predictors()).

There are a number of other steps included in the package:

#> [1] "step\_BoxCox" "step\_YeoJohnson"

#> [3] "step\_arrange" "step\_bagimpute"

#> [5] "step\_bin2factor" "step\_bs"

#> [7] "step\_center" "step\_classdist"

#> [9] "step\_corr" "step\_count"

#> [11] "step\_cut" "step\_date"

#> [13] "step\_depth" "step\_discretize"

#> [15] "step\_downsample" "step\_dummy"

#> [17] "step\_dummy\_multi\_choice" "step\_factor2string"

#> [19] "step\_filter" "step\_geodist"

#> [21] "step\_harmonic" "step\_holiday"

#> [23] "step\_hyperbolic" "step\_ica"

#> [25] "step\_impute\_bag" "step\_impute\_knn"

#> [27] "step\_impute\_linear" "step\_impute\_lower"

#> [29] "step\_impute\_mean" "step\_impute\_median"

#> [31] "step\_impute\_mode" "step\_impute\_roll"

#> [33] "step\_indicate\_na" "step\_integer"

#> [35] "step\_interact" "step\_intercept"

#> [37] "step\_inverse" "step\_invlogit"

#> [39] "step\_isomap" "step\_knnimpute"

#> [41] "step\_kpca" "step\_kpca\_poly"

#> [43] "step\_kpca\_rbf" "step\_lag"

#> [45] "step\_lincomb" "step\_log"

#> [47] "step\_logit" "step\_lowerimpute"

#> [49] "step\_meanimpute" "step\_medianimpute"

#> [51] "step\_modeimpute" "step\_mutate"

#> [53] "step\_mutate\_at" "step\_naomit"

#> [55] "step\_nnmf" "step\_normalize"

#> [57] "step\_novel" "step\_ns"

#> [59] "step\_num2factor" "step\_nzv"

#> [61] "step\_ordinalscore" "step\_other"

#> [63] "step\_pca" "step\_pls"

#> [65] "step\_poly" "step\_profile"

#> [67] "step\_range" "step\_ratio"

#> [69] "step\_regex" "step\_relevel"

#> [71] "step\_relu" "step\_rename"

#> [73] "step\_rename\_at" "step\_rm"

#> [75] "step\_rollimpute" "step\_sample"

#> [77] "step\_scale" "step\_select"

#> [79] "step\_shuffle" "step\_slice"

#> [81] "step\_spatialsign" "step\_sqrt"

#> [83] "step\_string2factor" "step\_unknown"

#> [85] "step\_unorder" "step\_upsample"

#> [87] "step\_window" "step\_zv"

**Checks**

Another type of operation that can be added to a recipes is a *check*. Checks conduct some sort of data validation and, if no issue is found, returns the data as-is; otherwise, an error is thrown.

For example, check\_missing will fail if any of the variables selected for validation have missing values. This check is done when the recipe is prepared as well as when any data are baked. Checks are added in the same way as steps:

trained\_rec <- trained\_rec %>%

check\_missing(contains("Marital"))

Currently, recipes includes:

#> [1] "check\_class" "check\_cols" "check\_missing" "check\_name"

#> [5] "check\_new\_values" "check\_range" "check\_type"

**S3 classes in recipes**

Lets build a very simple recipe:

library(tidyverse)

library(recipes)

rec <- recipe(mtcars) %>%

step\_center(everything()) %>%

step\_scale(everything()) %>%

check\_missing(everything())

rec %>% class()

## [1] "recipe"

As mentioned a recipe is a container for the steps and checks. It is a list of class recipe on which the prep, bake, print and tidy methods do the work as described in their respective documentation files. The steps and checks added to the recipe are stored inside this list. As you can see below, each of them are of their own subclass, as well as of the generic step or check classes.

rec$steps %>% map(class)

## [[1]]

## [1] "step\_center" "step"

##

## [[2]]

## [1] "step\_scale" "step"

##

## [[3]]

## [1] "check\_missing" "check"

Each subclass has the same four methods defined as the recipe class. As one of the methods is called on the recipe, it will call the same method on all the steps and checks that are added to the recipe. (Exception is prep.recipe, which does not only callprep on all the steps and checks, but also bake). This means that when implementing a new step or check, you should provide these four methods. Additionally, we’ll need the function that is called by the user to add it to the recipe object and a constructor .

**The recipes for recipes**

When writing a new step or check you will probably be inclined to copy-paste an existing step and start tweaking it from the top. Thus, first writing the function, then the constructor, and then the methods one by one. I think this is suboptimal and can get messy quickly. My preferred way is to start by not bothering about recipes at all, but write a general function that does the preparation work on a single vector. Once you are happy with this function you sit and think about which arguments to this function should be provided with upfront. These should be added as arguments to the function that is called by the user. Next you think about which function arguments are statistics that should be learned on the train set in the prep part of the recipe. You’ll then go on and do the constructor that is called by both the main function and the prep method. Only then you’ll write the function that is called by the user. You’ll custom step or check is completed by writing the four methods, since the functionality is already created, these are mostly bookkeeping.

For both checks and steps I made a skeleton available, in which all the “overhead” that should be in a new step or check is present. This is more convenient than copy-paste and existing example and try to figure out what is step specific and should be erased.

**Putting it to practice**

We are going to do two examples in which the *recipe for recipes* is applied.

**Example 1: A signed log**

First up is a signed-log, which is taking the log over the absolute value of a variable, multiplied by its original sign. Thus, enabling us to take logs over negative values. If a variable is between -1 and 1, we’ll set it to 0, otherwise things get messy.

**1) preparing the function**

This is what the function on a vector could look like, if we did not bother about recipes:

signed\_log <- function(x, base = exp(1)) {

ifelse(abs(x) < 1,

0,

sign(x) \* log(abs(x), base = base))

}

**2) think about the arguments**

The only argument of the function is base, that should be provided upfront when adding the step to a recipe object. There are no statistics to be learned in the prep step.

**3) the constructor**

Now we are going to write the first of the recipes functions. This is the constructor that produces new instances of the object of class step\_signed\_log. The first four arguments are present in each step or check, they are therefore part of the skeletons. The terms argument will hold the information on which columns the step should be performed. For role, train and skip, please see the documentation in one of the skeletons. base is step\_signed\_log specific, as used in 1). In prep.step\_signed\_log the tidyselect arguments will be converted to a character vector holding the actual column names. columns will store these names in the step\_signed\_log object. This container argument will not be necessary if the columns names are also present in another way. For instance, step\_center has the argument means, that will be populated by the means of the variables of the train set by its prep method. In the bake method the names of the columns to be prepared are already provided by the names of the means and there is need to use the columns argument.

step\_signed\_log\_new <-

function(terms = NULL,

role = NA,

skip = FALSE,

trained = FALSE,

base = NULL,

columns = NULL) {

step(

subclass = "signed\_log",

terms = terms,

role = role,

skip = skip,

trained = trained,

base = base,

columns = columns

)

}

**4) the function to add it to the recipe**

Next up is the function that will be called by the user to add the step to its recipe. You’ll see the internal helper function add\_step is called. It will expand the recipe with the step\_signed\_log object that is produced by the constructor we just created.

step\_signed\_log <-

function(recipe,

...,

role = NA,

skip = FALSE,

trained = FALSE,

base = exp(1),

columns = NULL) {

add\_step(

recipe,

step\_signed\_log\_new(

terms = ellipse\_check(...),

role = role,

skip = skip,

trained = trained,

base = base,

columns = columns

)

)

}

**5) the prep method**

As recognized in 2) we don’t have to do much in the prep method of this particular step, since the preparation of new sets does not depend on statistics learned on the train set. The only thing we do here is applying the internal function terms\_select function to replace the tidyselect selections, by the actual names of the columns on which step\_signed\_log should be applied. We call the constructor again, indicating that the step is trained and we supplying the column names at the columns argument.

prep.step\_signed\_log <- function(x,

training,

info = NULL,

...) {

col\_names <- terms\_select(x$terms, info = info)

step\_signed\_log\_new(

terms = x$terms,

role = x$role,

skip = x$skip,

trained = TRUE,

base = x$base,

columns = col\_names

)

}

**6) the bake method**

We are now ready to apply the baking function, designed at 1), inside the recipes framework. We loop through the variables, apply the function and return the updated data frame.

bake.step\_signed\_log <- function(object,

newdata,

...) {

col\_names <- object$columns

for (i in seq\_along(col\_names)) {

col <- newdata[[ col\_names[i] ]]

newdata[, col\_names[i]] <-

ifelse(abs(col) < 1,

0,

sign(col) \* log(abs(col), base = object$base))

}

as\_tibble(newdata)

}

**7) the print method**

This assures pretty printing of the recipe object to which step\_signed\_log is added. You use the internal printer function with a message specific for the step.

print.step\_signed\_log <-

function(x, width = max(20, options()$width - 30), ...) {

cat("Taking the signed log for ", sep = "")

printer(x$columns, x$terms, x$trained, width = width)

invisible(x)

}

**8) the tidy method**

Finally, tidy will add a line for this step to the data frame when the tidy method is called on a recipe.

tidy.step\_signed\_log <- function(x, ...) {

if (is\_trained(x)) {

res <- tibble(terms = x$columns)

} else {

res <- tibble(terms = sel2char(x$terms))

}

res

}

Lets do a quick check to see if it works as expected

recipe(data\_frame(x = 1)) %>%

step\_signed\_log(x) %>%

prep() %>%

bake(data\_frame(x = -3:3))

## # A tibble: 7 x 1

## x

##

## 1 -1.0986123

## 2 -0.6931472

## 3 0.0000000

## 4 0.0000000

## 5 0.0000000

## 6 0.6931472

## 7 1.0986123

**Example 2: A range check**

Model predictions might be invalid when the range of a variable in new data is shifted from the range of the variable in the train set. Lets do a second example in which we check if the range of a numeric variable is approximately equal to the range of the variable in the train set. We do so by checking if the variable’s minimum value in the new data is not smaller than its minimum value in the train set. The variable’s maximum value in the test set should not exceed the maximum in the train set. We allow for some slack (proportion of the variable range in the train set) to account for natural variation.

**1) preparing the function**

As mentioned, checks are about throwing informative errors if assumptions are not met. This is a function we could apply on new variables, without bothering about recipes:

range\_check\_func <- function(x,

lower,

upper,

slack\_prop = 0.05,

colname = "x") {

min\_x <- min(x)

max\_x <- max(x)

slack <- (upper - lower) \* slack\_prop

if (min\_x < (lower - slack) & max\_x > (upper + slack)) {

stop("min ", colname, " is ", min\_x, ", lower bound is ", lower - slack,

"\n", "max x is ", max\_x, ", upper bound is ", upper + slack,

call. = FALSE)

} else if (min\_x < (lower - slack)) {

stop("min ", colname, " is ", min\_x, ", lower bound is ", lower - slack,

call. = FALSE)

} else if (max\_x > (upper + slack)) {

stop("max ", colname, " is ", max\_x, ", upper bound is ", upper + slack,

call. = FALSE)

}

}

**2) think about the arguments**

The slack\_prop is a choice that the user of the check should make upfront. This is thus an argument of check\_range. Then there are two statistics to be learned in the prep method: lower and upper. These should be arguments of the function and the constructor as well. However, when calling the function these are always NULL, they are container arguments that will filled when calling prep.check\_range.

**3) the constructor**

We start again with the four arguments present in every step or check. Subsequently we add the three arguments that we recognized to be part of the check.

check\_range\_new <-

function(terms = NULL,

role = NA,

skip = FALSE,

trained = FALSE,

lower = NULL,

upper = NULL,

slack\_prop = NULL) {

check(subclass = "range",

terms = terms,

role = role,

trained = trained,

lower = lower,

upper = upper,

slack\_prop = slack\_prop)

}

**4) the function to add it to the recipe**

As we know by now, it is just calling the constructor and adding it to the recipe.

check\_range <-

function(recipe,

...,

role = NA,

skip = FALSE,

trained = FALSE,

lower = NULL,

upper = NULL,

slack\_prop = 0.05) {

add\_check(

recipe,

check\_range\_new(

terms = ellipse\_check(...),

role = role,

skip = skip,

trained = trained,

lower = lower,

upper = upper,

slack\_prop = slack\_prop

)

)

}

**5) the prep method**

Here the method is getting a lot more interesting, because we actually have work to do. We are calling vapply on each of the columns the check should be applied on, to derive the minimum and maximum. Again the constructor is called and the learned statistics are now populating the lower and upper arguments.

prep.check\_range <-

function(x,

training,

info = NULL,

...) {

col\_names <- terms\_select(x$terms, info = info)

lower\_vals <- vapply(training[ ,col\_names], min, c(min = 1),

na.rm = TRUE)

upper\_vals <- vapply(training[ ,col\_names], max, c(max = 1),

na.rm = TRUE)

check\_range\_new(

x$terms,

role = x$role,

trained = TRUE,

lower = lower\_vals,

upper = upper\_vals,

slack\_prop = x$slack\_prop

)

}

**6) the bake method**

The hard work has been done already. We just get the columns on which to apply the check and check them with the function we created at 1).

bake.check\_range <- function(object,

newdata,

...) {

col\_names <- names(object$lower)

for (i in seq\_along(col\_names)) {

colname <- col\_names[i]

range\_check\_func(newdata[[ colname ]],

object$lower[colname],

object$upper[colname],

object$slack\_prop,

colname)

}

as\_tibble(newdata)

}

**7) the print method**

print.check\_range <-

function(x, width = max(20, options()$width - 30), ...) {

cat("Checking range of ", sep = "")

printer(names(x$lower), x$terms, x$trained, width = width)

invisible(x)

}

**8) the tidy method**

tidy.check\_range <- function(x, ...) {

if (is\_trained(x)) {

res <- tibble(terms = x$columns)

} else {

res <- tibble(terms = sel2char(x$terms))

}

res

}

Again, we check quickly if it works

cr1 <- data\_frame(x = 0:100)

cr2 <- data\_frame(x = -1:101)

cr3 <- data\_frame(x = -6:100)

cr4 <- data\_frame(x = 0:106)

recipe\_cr <- recipe(cr1) %>% check\_range(x) %>% prep()

cr2\_baked <- recipe\_cr %>% bake(cr2)

cr3\_baked <- recipe\_cr %>% bake(cr3)

## Error: min x is -6, lower bound is -5

cr4\_baked <- recipe\_cr %>% bake(cr4)

## Error: max x is 106, upper bound is 105

**Conclusion**

If you like to add your own data preparation steps and data checks to the recipes package, I advise you to do this in a structured way so you are not distracting by the bookkeeping while implementing the functionality. I propose eight subsequent parts to develop a new step or check:

1) Create a function on a vector that could be applied in the bake method, but does not bother about recipes yet.  
2) Recognize which function arguments should be provided upfront and which should be learned in the prep method.  
3) Create a constructor in the form step\_\_new or check\_\_new.  
4) Create the actual function to add the step or check to a recipe, in the form step\_ or check\_.  
5) Write the prep method.  
6) Write the bake method.  
7) Write the print method.  
8) Write the tidy method.