```
knitr::opts_chunk$set(root.dir =
("home/westland/Desktoop/aaa_chapters_FINAL/"))
```

'Test Decks' and their progeny

In the early days of IBM 360 accounting systems (IBM's advertising emphasized their 360-degree-full-circle-of-service) transactions were input on decks of 80-column Hollerith cards. Auditors would test automated controls by reading a deck of known transactions (i.e., a 'test deck') and comparing these with the accounting reports generated by a client's automated system. In those days, this was termed 'auditing *around* the computer. Auditors contrasted this with 'auditing *through* the computer' which followed transaction processing through the internal computer operations in order to verify that controls were functioning properly. It is hard for me to believe that any audit budget could survive auditors tracing a transaction's journey through the code, yet this seemed to have been part of that period's vernacular.

Though Hollerith cards are rarely seen these days outside a computer museum or eBay 'vintage artifact' listing, the concept of testing a client's automated 'black-box' (i.e., a system where access to the code is limited by time, effort or ownership). Simulations allow the auditor to compare known transaction inputs to computed output. Modern computing systems rely on code that is proprietary, with ownership by someone other than the client when systems are hosted on service bureaus and cloud platforms. Simulation of accounting transactions with known distributions and error occurances provide useful tools for audit testing of black boxes. The current chapter details methods for creating simulated accounting transactions with known distribution and error characteristics.

Setting up the simulation

Whatever the purpose, it is essential to generate realistic, large, replicable audit data. This chapter provides annotated code for realistically distributed accounting data. Our boilerplate example focuses on the two accounting cycles that generate the majority transactions analyzed in an audit – the Sales & collections cycle and the Procurement & payment cycle. These cycles are the main contributors to accounts for sales, cost of goods sold, cash, inventory, accounts receivable and working capital.

With a few exceptions, all of our code uses R-base commands. The exceptions are Dirk Eddelbuettel’s ‘random’ package; Hadley Wickham’s ‘tidyverse’ and ‘plyr’ packages; and Vitalie Spinu’s ‘lubridate’ package for date-time formatting and arithmetic. Although they are used minimally in the subsequent code, they are three widely used packages in data science. The reader should become familiar with these packages as they make analysis and manipulation of accounting data simple and reliable.

I have used the “file$variable” format rather than attaching and detaching files, which creates less confusion when you have many files. Variable names are descriptive, all lower case, connected by underlines – Hadley Wickham calls this “snake_case.”

Assumptions used in the generation of simulated data

The following assumptions are made in generating the accounting transaction simulations.

Document-updating events in the accounting cycle all have: identifier, date, number of inventory items, unit value

1. Entities

- stock_on_hand
- customer
- vendor
- bank

Events associated with entities

1. Sales cycle (amounts at sales price)

- sale
- sale_return
- shipment
- cash _receipt (cash sale)
- account_receivable (credit sale)
- collection (credit sale)
- deposit (bank)

Procurement cycle (to replace sold inventory; amounts at cost)

- inventory
- purchase_order
- account_payable
- receivers (for ordered inventory)
- disbursement (in payment to retire account_payable))

Reporting

- Balance Sheet
- Income Statement
- Statement of Changes in Working Capital
- Statement of Changes in Retained Earnings

Assumption: there is no inflation in either buyer or seller markets throughout the year.

Assumption: documents are not grouped, rather one preceding transactions (e.g., Sales) triggers one and only one succeeding transaction (e.g., AR)

Document Generation

Accounting transaction information is highly multicolinear because many journal entries are triggered by other transactions. The subsystems simulated here are summarized in an influence chart that shows the cascade of triggered transactions.

```
library(DiagrammeR)


grViz("
digraph boxes_and_circles {

  # a 'graph' statement
  graph [overlap = true, fontsize = 10]

  #  'node' statements
  node [shape = box,
        fontname = Helvetica]
  Sales; AR; Collections; Bank; Inventory; PO; AP; Receivers;
Disbursements; Expenses

  # 'edge' statements

 Sales -> AR
 AR -> Collections
 Sales -> Bank
 Collections -> Bank
 Sales -> Inventory
 Inventory -> PO
 PO -> AP
 PO -> Receivers
 Receivers -> Disbursements
 Disbursements ->  Bank
 Expenses ->  Bank

}
")
```

Transactions are generated in the following order (items in ${}$ means transfer data from other documents)

- sale_trigger event = {perpetual_count = perpetual_count - sale_count}
- inventory_reorder trigger event = {inventory/perpetual_count< reorder_count}
- inventory sku, unit_cost, unit sales price, count_beginning, reorder_count {= constant * sqrt(annual_sales)}
- sale invoice_no, invoice_date, sku, sale_count, sale_unit_price, sale_extended cash_or_ar, sale_return, perpetual_count {updated on sale_trigger} , cash_receipt {sale sub-classification}
- account_receivable {sale sub-classification}
- sale_return {sale sub-classification}
- shipment shipper_no, customer_no, invoice_no, shipper_date = {invoice_date + delay}

- collection receipt_no, receipt_amount = {sale_amount * unit_price}, receipt_date = {invoice_date + delay} or {invoice_date} for cash
- deposit deposit_no, deposit_amount = {sale_amount * unit_price}, deposit_date = {receipt_date + delay}
- purchase_order po_number, po_date, unit cost, reorder_amount (created on inventory_reorder trigger)
- receipt receiver_no, receiver_count, reorder_amount, receiver_date = {po_date + delay}
- account_payable ap_no, receiver_no, unit cost, reorder_count, ap_date = {receiver_date}
- disbursement disb_no, disb_amount, disb_date = {receiver_date + delay}

Strategy for Document Generation

1. The simulation creates two main files that represent starts of cascades of triggering events \$-\$ (1) the *sales and inventory reduction* file and (2) the *procurement and inventory replenishment* file. Client's documents will be generated as a final step
2. Amounts (prices, costs, counts) will be generated with a Poisson random number generator (positive integers)
3. Delays will be generated with an exponential random number generator (positive reals)
4. Classifiers will be generated using a Bernoulli random number generator (0/1)
5. Identifiers will be generated with sequential numbers ordered by date, with a transaction type prefix; e.g., s000123 will be a sale, with prefix ‘s’

Statistical assumptions and the distribution of accounting transactions

When one wishes to infer conclusions from some set of data – which is the objective of an audit – the data are a priori assumed to follow some probability distribution. Choice of the correct distribution for a specific situation or account is one of the most important aspects of effective inference from data, and predicates efficient and effective auditing.

In addition to choosing the right distribution, auditors need to determine how much data to collect – the appropriate sample size that balances the cost of an incorrect decision with the cost of auditing. Data collection and processing in auditing is expensive – audit costs for a single sampled transaction may run into hundreds or thousands of dollars. The total cost of an audit is extremely sensitive to the right choice of distributions, sample sizes, methodology for inference and decision models. Auditing that is cavalier with methods, relying too much on personal inference, will drive up audit costs and raise the probability of an incorrect opinion. Inferential shortcomings have created huge and embarrassing failures for audit firms over the past two decades – such failures drove Arthur Andersen into bankruptcy.

Sample size determination is one of the most confusing areas of statistics. Ideally we would not choose a sample size ‘up front,’ but would continue sampling until a particular objective is met (e.g., until we determine that the system is either in or out of control). This is called ‘optimal stopping’ and is often used in laboratory and factory sampling. Where we need to do a ‘one shot’ sample, we need to know the distribution of the parameter we are estimating – this is usually impossible to do accurately, but that doesn’t stop us from trying.

Many auditors assume they can simply assumed client transactions to be Normally distributed, perhaps invoking but not understanding the Central Limit Theorem. This is usually a very bad assumption in auditing, since all audited distributions should be censored at zero and are usually right skewed and highly kurtotic. It is also not uncommon to find that transaction distributions are multi-modal (Westland, 2017). Furthermore, attributes will have a discrete distribution, often with a limited set of 'valid' values

Thus auditor cannot assume that 'one-size-fits-all' in sample size determination. There are three factors that need to be considered:

1. effect size: for attribute error rates this is 'tolerable error'; for value estimation this is the proportion of materiality assigned to an account, also called 'tolerable error'.

2. population distribution and parameters: A generally safe assumption is that the attribute errors or the dollar errors are Poisson distributed. When transactions are Poisson distributed, then the waiting time between transactions are typically be assumed to be distributed exponentially when the transactions are independent. Compliance and substantive errors in transactions and financial accounts are relatively rare events – if they were not then financial statements would be unusable and the scope of auditing would be unmanageable (a situation sometimes found in financial fraud cases). The Law of Rare Events supports an assumption of Poisson distribution for monetary balances, and an exponential distribution for delays; e.g., the time a customer takes to pay their account receivable. The Poisson distribution is also sometimes called the law of small numbers (after the title of an 1896 book by Ladislaus Bortkiewicz) because it is the probability distribution of the number of occurrences of an event that happens rarely but has very many opportunities to happen. Numerous studies support the assumption of a Poisson process generating the distribution of accounting errors with (Ball & Watts, 2012; Fienberg, Neter, & Leitch, 1977; Garstka, 1977; Greene, 1994; Keenan & Kotula, 2004; Repin, Sauter, & Smolianski, 2004).

3. confidence in our decision that: e.g.,that the error rate observed in the sample would cause us to change our audit decision, or an investor to change their investment decision. Power and significance of audit tests set our formal definition of 'confidence.' Computing sample size here is analogous to standard univariate calculations (Cochran 1977; Kish 1955; Lohr 1999; Snedecor and Cochran 1989, Westland and See-to 2007) but using a formulation for variance customized to this problem.

General Parameters of the Simulation

Based on our prior assumptions, we start by setting the simulation-wide parameters. These will include the transaction volumes, markups of costs to sales values, numbers of customers, inventory items, and so forth. Any of these parameters, as well as certain other parameters in the following code can be customized to the user's own needs.

```
###########################################################################
####
##
## 1 - Simulation-wide parameters and general use functions
##
###########################################################################
####
```

```
## clear workspace (recommended)

rm(list=ls())

##
##  your seed number is the starting point used in the
##  generation of a sequence of random numbers
##  assuring that you will obtain the same
##  accounting datasets (given the same seed number)
##  when you rerun the code
##

set.seed(123456)  ## setting the seed assures that we can recreate the
dataset by rerunning the code

#
#  Fiscal year beginning and end dates
#

fyear_begin <- paste0(format(Sys.Date(), '%Y'), '-01-01')  # date can be
hard-coded if needed
fyear_end <- paste0(format(Sys.Date(), '%Y'), '-12-31')

#
# Inventory related parameters
#
no_of_sku <- 10     ## number stock-keepting-units (sku), i.e., distinct
inventory items
lngth_of_sku <- 5   ## number of alphanumeric characters in sku identifier
avg_cost <- 1000    ## ave. cost of sku from vendor
avg_markup <- 200   ## ave. markup of sales_price sku from vendor
avg_count <- 300    ## ave. number of items of sku
#
# Sales related Parameters
#
no_of_cust = 10     ## number of customer for sales
no_of_sales = 1000      ## number of sales transactions
avg_count_sold = 15     ## average number of items sold per transaction
#
#  Shipping and A/R collection related parameters
#  Starting balances have a date field of "1900-01-01" to avoid non-date
contents
#
days_to_ship <- 10              ## average of 10 days to ship
days_to_collect_ar <- 60        ## average of 60 days to collect on A/R,
percent_ar_unpaid <- .02        ## 2% of AR are never paid (in default)
ar_in_default_cutoff <- 500     ## over 499 days late is in default
#
#  Purchase related parameters
#
days_to_receive_inventory <- 15  # average of 15 days to receive ordered
inventory
ap_record_created_delay <- 1     # account payable record created 1 day
after receiver of inventory
```

```
ap_disbursement_delay <- 75        # account payable payment disbursement
on average 75 days after receiver of inventory
#
##
## random dates
##

starting_inventory_budget <- 1000 # the approximate $ inventory on hand at
the start of the simulation


rdate <- function(x,
                  min = paste0(format(Sys.Date(), '%Y'), '-01-01'),
                  max = paste0(format(Sys.Date(), '%Y'), '-12-31'),
                  sort = TRUE) {
    dates <- sample(seq(as.Date(min), as.Date(max), by = "day"), x, replace
= TRUE)
    if (sort == TRUE) {
        sort(dates)
    } else {
        dates
    }
}
```

This last function helps in generating realistic dates and time intervals for the simulation

The Sales Journal

Assuming we have inventory to sell, our transaction recording begins when a customer takes delivery of inventory in exchange for cash or an obligation to pay (i.e., and account receivable). The individual sales transactions are recorded in a sales journal. In the past, ‘journals’ were physical books, where the original occurrence of a transaction was recorded, along with ad hoc notes indicating the nature of the transaction. ‘Ledgers’ on the other hand were physical books that were used to track inventory and other stock items. Today, records are digital and descriptive fields are strictly defined in advance.
The following code generates the original transaction from a sale to a customer, as well as those transactions that are triggered by a subsequent event (e.g., collection) related to that particular sales transaction.

```


##############################################
# 2 – Populate the Sales Journal file
##############################################


library(tidyverse)
library(lubridate)
library(random)

#
#   Create a sales_journal and then populate it
#   Sales prefix = “i”; invoice_no is sequenced by date)
#   order by invoice date (use lubridate and plyr)
#

invoice_no <- seq(from = 1, to = no_of_sales)
invoice_no <- sprintf("i%05d", invoice_no)

invoice_date <- ymd(rdate(no_of_sales, min = fyear_begin, max =
fyear_end))

sales_journal <-
  data.frame(invoice_no,invoice_date) %>%
    arrange(invoice_date)

#
#  create customers and tie them to sales
#

customer_string <- seq(from = 1, to = no_of_cust)
customer_string <- sprintf("c%05d", customer_string)
customer_no = sample(customer_string, no_of_sales, replace = T,
prob=runif(no_of_cust,0,1))
sales_journal = data.frame(customer_no, sales_journal)

#
#  create skus and their vendors
#

sku <- randomStrings(n=no_of_sku, len=lngth_of_sku, digits=F,
upperalpha=TRUE,
                     loweralpha=F, unique=TRUE, check=TRUE)
vendor_string <- seq(from = 1, to = no_of_sku / 3)
vendor_string <- sprintf("v%05d", vendor_string)
vendor_no = sample(vendor_string, no_of_sku, replace = T,
prob=runif(no_of_sku/3,0,1))
sku = data.frame(vendor_no, sku) %>%
  rename(sku = V1)


sales_count <- rpois(no_of_sales,avg_count_sold)
sku_of_sale <- sample(sku$sku, no_of_sales, replace=TRUE)  ## sku from 1.1
sales_return <-  rbinom(no_of_sales, 1, .05)   ## 1 if there was a return
on this sale
cash_not_ar <- rbinom(no_of_sales, 1, .2)   ## 1 is a cash sale (20%), 0 is
credit (80%)

unit_cost <- rpois(nrow(sku), 30)
sales_unit_price <-  unit_cost * ceiling(abs(rnorm(nrow(sku), 2, 1)))

#
#  Write sales_journal
#

sales_journal <- do.call(
    cbind.data.frame,
    list(sales_journal,
        sku_of_sale,
        sales_count,
        cash_not_ar,
        sales_return,
        unit_cost,
        sales_unit_price
        )
)


colnames(sales_journal) <-
  c("customer_no",
    "invoice_no",
    "invoice_date",
    "sku",
    "sales_count",
    "cash_not_ar",
    "sales_return",
    "unit_cost",
    "sales_unit_price")
```

```


##########################################################
# 3 – Add inventory and other information to Sales Journal
##      shipments & collections,
##      which are defined by unique IDs and date delays
##      2% of AR are never paid (in default)
##########################################################

library(lubridate)

sales_journal$sales_extended <- sales_journal$sales_count *
sales_journal$sales_unit_price

shipper_date  <-  invoice_date + round(rexp(no_of_sales,1/days_to_ship),0)


collection_amount <- sales_journal$sales_extended   ## set amount for cash
sales
collection_date <- sales_journal$invoice_date   ## set date for cash sales
for(i in 1:no_of_sales) {
    if(cash_not_ar[i] == 0) {  ## collect only on credit sales
        if(rbinom(1, 1, percent_ar_unpaid) == 1) {collection_date[i] <-
shipper_date[i] + ar_in_default_cutoff}  ## 1 is a default
(percent_ar_unpaid %)
        collection_date[i] <- shipper_date[i] +
round(rexp(1,1/days_to_collect_ar),0)
```

```
        collection_amount[i] <- sales_journal$sales_extended[i]
     }}


#
##   collection_no prefix = “r”;  is sequenced by date)
#    (lubridate and plyr)
#

sales_journal$collection_amount <- sales_journal$collection_amount <-
as.numeric(collection_amount)
sales_journal$collection_date <- sales_journal$collection_date <-
ymd(collection_date)
sales_journal <- arrange(sales_journal, collection_date)
sales_journal$collection_no <- sprintf("r%05d", seq(from = 1, to =
no_of_sales))

sales_journal <- arrange(sales_journal, invoice_date)
sales_journal <- arrange(sales_journal, invoice_date)
sales_journal$collection_no <- sales_journal$collection_no

#
##   shipper_no prefix = “s”;  is sequenced by date)
#    (use lubridate and plyr)
#
sales_journal$shipper_date <- ymd(shipper_date)
sales_journal <- arrange(sales_journal, shipper_date)
sales_journal$shipper_no <- sprintf("s%05d", seq(from = 1, to =
no_of_sales))


sales_journal <- arrange(sales_journal, invoice_date)
sales_journal <- arrange(sales_journal, invoice_date)
sales_journal$shipper_no <- sales_journal$shipper_no
sales_journal$shipper_date <- sales_journal$shipper_date


collections <- sales_journal %>%
  select(customer_no,
         invoice_date,
         invoice_no,
         sales_extended,
         collection_amount,
         collection_date,
         collection_no,
         cash_not_ar,
         sku)

collections <- split(collections, cash_not_ar)$'0'   ## credit only


shipments_journal <-
  sales_journal %>%
  select(shipper_no, date=shipper_date, customer_no, sku, invoice_no,
invoice_date, quantity = sales_count)
```

Cash and Bank Deposits

Cash sales and collections on accounts receivable generate cash, which is deposited to the bank. Inventory procurement and other expenditures will use cash from the bank account. We assume that deposits are made daily in this code.

```
##########################################################
# 4 – Generate daily bank deposit records
##
##########################################################

library(tidyverse)

##
## Daily deposits
##

deposit_daily <-
  sales_journal %>%
  group_by(invoice_date) %>%
  summarize(deposit_amount = sum(sales_extended)) %>%
  as.data.frame()

deposit_no <- sprintf("d%05d", seq(from = 1, to = nrow(deposit_daily)))
deposit_daily <-
  cbind(deposit_daily, deposit_no) %>%
  rename(deposit_date = invoice_date)
```

Inventory and Purchase Orders for Inventory Replenishment

Eventually after selling enough inventory, that inventory will need to be replenished. In practice, inventory levels for any particular stock keeping unit (SKU) are monitored, and a purchase order is generated when inventory falls below a certain point. In this code we used a standard economic order quantity model to determine when a purchase order is triggered. The purchase order is sent to the vendor, who after a certain ‘lead time’ delivers the inventory to your client. Delivery is recorded as a ‘receipt’ and a receiver (transaction document) is written up, as well as an account payable (voucher).

Generators have been an important part of Python ever since they were introduced with PEP 255. Generator functions allow you to declare a function that behaves like an iterator, but in a fast, easy, and clean way. An iterator is an object that can be iterated (looped) upon. It is used to abstract a container of data to make it behave like an iterable object. Simply speaking, a generator is a function that returns an object (iterator) which we can iterate over (one value at a time).

A generator function is a special type of function that you call repeatedly to obtain a sequence of values. Often, generators need to maintain internal state, so they're typically constructed by calling another function that returns the generator function (the environment of the function that returns the generator is then used to track state). For example, the following sequence_generator() function returns a generator function that yields an infinite sequence of numbers:

```
sequence_generator <- function(start) {
value <- start - 1
function() {
value <<- value + 1
value
}
}
> gen <- sequence_generator(10)
> gen()  # 10
> gen()  # 11
```

Generator functions can signal completion by returning the value NULL.

The current state of the generator is the value variable that's defined outside of the function. Note that superassignment (<<-) operator is used to update this state from within the function. The good use of superassignment is in conjuction with lexical scope, where an environment stores state for a function or set of functions that modify the state by using superassignment. The reason that this works in Python is because of the $yield$ keyword. This causes the loop in the function to give control back to whatever called the function. This means that we can call next(gen) without having to worry about an infinite about of numbers filling up the memory on the machine.

My simulation will use a fixed 50 unit order quantity, and fixed 50 unit reorder point for generation of purchase orders. A more sophisticated approach would use economic order quantities (EOQ) to set the size of purchase orders; this could be implemented via R's *plyr* package as shown below.

```



##########################################################
# 5 – Generate inventory EOQs (economic order quantities)
##
##########################################################

library(plyr)
library(kableExtra)

###
## EOQ = sqrt(2SD / H) where S = Setup costs, D = Demand rate, H = Holding
costs
##    so reorder_count should be proportional to sqrt(D)
###

reorder <-
```

```
  ddply(sales_journal, .(sku), summarize,
        reorder=round((sum(sales_count))^.5,0))

kable(reorder, caption = "Economic Order Quantities for Each SKU") %>%
  kable_styling(bootstrap_options = "striped")
```

Inventory

Sales and procurement cycles represent activities that consume and replenish inventory. Thus our starting point for simulation is inventory – its identifiers, vendors, pricing, and so forth.

```
reorder_stock_level <- 50
  reorder_count <-  100

##############################################
# 6– Generate the Inventory (by SKU) file
##############################################

library(lubridate)
library(tidyverse)

##  GENERATOR FUNCTIONS FOR THE SEQUENTIAL TRANSACTION NUMBERING
#
##   Purchase order prefix = “p”;
##   voucher_no prefix = “ap”;
##   disbursement_no prefix = “dis”;
##   receiver_no prefix = “rec”;
#

po_no_generator <- function(){
  i <- 0
  function(){
    i <<- i + 1
    sprintf("p%05d", i)
  }
}
new_po <- po_no_generator()

rec_no_generator <- function(){
  i <- 0
  function(){
    i <<- i + 1
    sprintf("rec%05d", i)
  }
}
new_rec <- rec_no_generator()

ap_no_generator <- function(){
  i <- 0
```

```
    function(){
      i <<- i + 1
      sprintf("ap%05d", i)
    }
  }
  new_ap <- ap_no_generator()

  disb_no_generator <- function(){
    i <- 0
    function(){
      i <<- i + 1
      sprintf("dis%05d", i)
    }
  }
  new_disb <- disb_no_generator()

  ## compute the starting inventory for each stock-keeping unit (sku)
  ave_starting_inventory <- 100

  fyear_begin_inventory_ledger <-
    sales_journal %>%
    as.data.frame() %>%
    group_by(sku) %>%
    slice(1) %>%
    select(sku, unit_cost, sales_unit_price) %>%
    as.data.frame() %>%
    mutate(stock_on_hand = ave_starting_inventory +
  floor(rnorm(no_of_sku,0,10)),
           invoice_no=0,
           invoice_date=as_date("1900-01-01"),
           sales_count=0,
           sales_return=0
    )



  #######################################
  ##
  ##  Create the purchase journal
  ##   based on the reorder_stock_level &
  ##   reorder_count for each sku inventory
  ##
  ######################################

  sku_stock <-
    fyear_begin_inventory_ledger %>%
    select(sku, stock_on_hand)

  reorder_stock_level <- 50
    reorder_count <-  100



  for(i in 1:nrow(sales_journal)){
    for(j in 1:nrow(sku_stock))
```

```
   if(sku_stock$sku[j] == sales_journal$sku[i]){

     sales_journal$running[i] <-
       sku_stock$stock_on_hand[j] -
       sales_journal$sales_count[i] +
            sales_journal$sales_return[i]

     sku_stock$stock_on_hand[j] <-
       ifelse(sales_journal$running[i] < reorder_stock_level,
       sales_journal$running[i] + reorder_count,
       sales_journal$running[i])

     sales_journal$po_count[i] <-
       ifelse(sales_journal$running[i] < reorder_stock_level,
              reorder_count,
              0)
     sales_journal$po_no[i]   <-
       ifelse(sales_journal$running[i] < reorder_stock_level,
              new_po(),
              0)
     sales_journal$receiver_no[i]     <-
       ifelse(sales_journal$running[i] < reorder_stock_level,
              new_rec(),
              0)

     sales_journal$ap_no[i]     <-
       ifelse(sales_journal$running[i] < reorder_stock_level,
              new_ap(),
              0)
     sales_journal$disbursement_no[i]    <-
       ifelse(sales_journal$running[i] < reorder_stock_level,
              new_disb(),
              0)

     ## note that ifelse fails to parse dates, and
     ## I use a difference approach for the dates that doesn't use 'if'
     ## rather it prunes out the non-PO rows to create the purchase journal


     ## Also note that I am updating inventory on the po_date, but the
correct
     ## date will be the receiver_date.  This will create inventory count
(cutoff)
     ## discrepancies; the number of these can be modulated
     ## by changing the days_to_receive_inventory parameter at the beginning
of the code

    sales_journal$receiver_date[i]  <-
       sales_journal$invoice_date[i] +
round(rexp(1,1/days_to_receive_inventory))

    sales_journal$po_date[i]  <-
        sales_journal$invoice_date[i]
     sales_journal$ap_date[i]  <-
               sales_journal$receiver_date[i]  + ap_record_created_delay

     sales_journal$disbursement_date[i]    <-
               sales_journal$receiver_date[i]  + round(rexp(1,1/75),0)


   }
 }

  sales_journal$receiver_date  <-  as_date(sales_journal$receiver_date)
  sales_journal$po_date <-  as_date(sales_journal$po_date)
  sales_journal$ap_date <-  as_date(sales_journal$ap_date)
  sales_journal$disbursement_date <-
 as_date(sales_journal$disbursement_date)

 ##################################################
 ##
 ##   Create the various purchase related journals
 ##
 #################################################

 purchase_journal <-
    sales_journal %>%
    filter(po_count != 0) %>%
    select(sku, po_count, unit_cost,
           po_no, po_date,
           receiver_no, receiver_date,
           ap_no, ap_date,
           disbursement_no, disbursement_date)

 perpetual_inventory_ledger <-
    sales_journal %>%
    arrange(sku,invoice_date) %>%
    select(sku, date = invoice_date, stock_on_hand = running)

 sales_journal <- sales_journal[,1:15]

 receiver_journal <-
    purchase_journal %>%
    select(receiver_no, receiver_date,
           sku, unit_cost, received = po_count, po_no, po_date)

 ap_ledger <-
    purchase_journal %>%
    mutate(extended_cost = po_count * unit_cost) %>%
    select(ap_no, ap_date, sku, no_units_ordered = po_count, extended_cost,
           receiver_no, receiver_date)

 disbursement_journal <-
    purchase_journal %>%
    select(disbursement_no, disbursement_date, sku,
           no_units_ordered = po_count,
           unit_cost,ap_no,ap_date)
```

```
purchase_journal <-
  purchase_journal %>%
   select(po_no, po_date,sku, po_count, unit_cost)


############################################################################
#
## Cost of Goods Sold
##  After sales, the most significant account on the income statement is
## cost of goods sold (essentially sales restated
## at their procurement / manufacturing cost).
############################################################################
###

daily_sales_cgs_margin <-
  sales_journal %>%
  group_by(invoice_date) %>%
  mutate(cgs = sum(sales_count * unit_cost),
         gross_margin = sum((sales_unit_price - unit_cost) * sales_count)
  )%>%
  summarize(total_sales = sum(sales_extended),
            cgs=sum(cgs),
            gross_margin = sum(gross_margin),
            date = first(invoice_date)) %>%
  ungroup() %>%
  as.data.frame() %>%
  select(date , total_sales, cgs, gross_margin)
```

Perpetual Inventory, Accounts Payable and Other Inventory Related Accounts

Before the computerization of accounting in the 1970s, most inventory tracking was performed with periodic counts. Cycle counts of subsets of SKUs would update the inventory ledger. There were two significant problems with periodic inventory: (1) stockouts and obsolescence were common due to the disconnect between transactions and actual balances; and (2) thefts and shortages were difficult to control. Computerization made perpetual (i.e., up to date) inventory balances possible, while reducing the labor cost associated with counts. By the 1990s, most companies were relying on perpetual inventory systems. A significant part of the audit is spent on validating the accuracy of perpetual inventory balances.

The prior code chunk computed inventory reorders, and the associated accounts in response to sales demand. I provide a code chunk here to break out the separate accounting documents that would appear in an accounting system.

The purchase order is sent to the vendor, who after a certain 'lead time' delivers the inventory to your client. Delivery is recorded as a 'receipt' and a receiver (transaction document) is written up, as well as an account payable (voucher). The term 'voucher' refers to an old paper system that was used throughout the

19th and 20th centuries, where paperwork from purchase orders, receivers, debit memos, disbursements and so forth were stapled together and filed. You will still find that term used in digital accounting systems.

Customer Credit Limits and Outstanding Accounts Receivable

The amount of credit extended to customers is typically monitored carefully to limit the risk of non-payment of accounts receivable. The risk of non-payment and suggestions for credit limits to particular customers may be obtained from service companies such as Dun & Bradstreet. Typically a customer is given a credit limit based on their payment history with the company as well as external assessments of risk.

```
#############################################################################
########
## 7 –  customers' credit limits
## cr limit proportional to 10 x {annual sales | sku} / {time_to_pay} +
{random}
## we would like to have some customers exceeding their credit limit
#############################################################################
########

library(tidyverse)
library(lubridate)
library(kableExtra)


cr_limit <-
  sales_journal %>%
  group_by(customer_no) %>%
  mutate(ann_sales = sum(sales_extended),
         time_to_pay = mean(as.numeric(collection_date - invoice_date)),
         credit_limit = 10 * round(ann_sales / time_to_pay, digits=-2)) %>%
  slice(1) %>%
  select(customer_no, credit_limit)



kable(cr_limit, caption = "Customer Credit Limits") %>%
  kable_styling(bootstrap_options = "striped")
```

Accounts Receivable

I assume that the business has just started this year. Were we to assume a beginning accounts receivable balance, I would also need to generate a set of sales transactions from the prior year, with shipments and collections, which would add complexity without any accompanying insights. This can be left as an excercise for the interested reader. The sales would be generated by replicating the mechanisms that I have already used to generate sales in the audit year.

Credit sales generate accounts receivable, which may (or may not) ultimately be collected, but also are sometimes left unpaid, for a variety of reasons. The confirmation of accounts receivable and assessment of uncollectable accounts receivable are consider two of the most important tasks in an audit.

```
############################################################################
##
#  8 - accounts receivable (updated daily)
##  follows the pattern to compute daily perpetual ar
##  you will need this for the tests of whether customers exceed credit
limits
############################################################################
##

library(tidyverse)
library(lubridate)

real_world_credit_sales <-
  credit_sales_journal <-
  sales_journal[,1:15] %>%
  filter(cash_not_ar == 0)   ## credit only

real_world_cash_sales <-
  cash_sales_journal <-
  sales_journal[,1:15] %>%
  filter(cash_not_ar == 1)     ## cash only


## put daily AR, plus an aging ...

fyear_dates <-  data.frame(f_date=seq(as.Date(fyear_begin),
as.Date(fyear_end), "day"))
# sales_journal <-  full_join(sales_journal,fyear_dates,
by=c("invoice_date"="f_date") )

#%>%  full_join(fyear_dates, by=c("date"="f_date"))

daily_cum_ar_balance <-
  sales_journal %>%
  filter(cash_not_ar == 0) %>%    ## credit sales only
  arrange(invoice_date) %>%
  group_by(invoice_date) %>%
  summarize(sales_sum = sum(sales_extended))%>%
  ungroup(invoice_date) %>%
  right_join(fyear_dates, by=c("invoice_date"="f_date"))

daily_cum_ar_balance[is.na(daily_cum_ar_balance)] <- 0

daily_cum_ar_balance <-
  daily_cum_ar_balance %>%
  mutate(ar_cum = cumsum(sales_sum))  %>%
  select(date=invoice_date, ar_cum) %>%
  arrange(date)

#daily_cum_ar_balance[is.na(daily_cum_ar_balance)] <- 0

daily_cum_collections <-
  sales_journal %>%
  filter(cash_not_ar == 0) %>%    ## credit sales only
  arrange(collection_date) %>%
  group_by(collection_date) %>%
  summarize(collect_sum = sum(collection_amount)) %>%
  ungroup(collection_date) %>%
  right_join(fyear_dates, by=c("collection_date"="f_date"))

daily_cum_collections[is.na(daily_cum_collections)] <- 0

daily_cum_collections <-
  daily_cum_collections %>%
  as.data.frame() %>%
  filter(collection_date <= fyear_end) %>%
  mutate(collect_cum = cumsum(collect_sum))  %>%
  select(date = collection_date, collect_cum) %>%
  arrange(date)

daily_ar_balance <- inner_join(daily_cum_ar_balance, daily_cum_collections,
"date")

daily_ar_balance <-
  daily_ar_balance %>%
  mutate(ar_balance = ar_cum - collect_cum) %>%
  select(date,ar_cum , collect_cum, ar_balance) %>%
  arrange(date)
```

Accounts Receivable Aging

The accounts receivable aging may be either automatically compiled by the client, as it is here; or the auditor may compile it from the workpapers, as would be done during substantive testing. The accounts receivable aging provides a basis for revising the amounts in the allowance for uncollectable accounts. Typically there are 'rules of thumb' for the collectability of accounts that have been outstanding and unpaid for a certain amount of time; this will vary by client and by customer. As was argued in chapter 9 on substantive testing, a more formal basis can be created for estimation and allowance for uncollectable accounts using time series forecasting. For this chapter, I simply print the aging that would be the basis for traditional ways of calculating the allowance for doubtful accounts.

```
###########################################################################
##
#  9 - accounts receivable  aging
##
```

```
###########################################################################
##


library(tidyverse)
library(lubridate)
library(kableExtra)

## create age breaks at <45 days; 45 - 120 days; >120 days

i_date <- yday(ymd(as_date(sales_journal$invoice_date)))
ye_date <- yday(ymd(as_date(fyear_end)))

age_ar <-
  sales_journal %>%
  filter(cash_not_ar == 0) %>%
  filter(collection_date > fyear_end) %>%
  mutate(i_date <- yday(ymd(as_date(invoice_date))),
         ye_date <- yday(ymd(as_date(fyear_end))),
         age =ye_date - i_date,
         age_lt_45 = ifelse(age<45, sales_extended, 0),
         age_45_120 = ifelse(age<=120 & age>=45,sales_extended,0),
         age_gt_120 = ifelse(age>120,sales_extended,0)   ## redundant but
easier to read
         ) %>%
  select(customer_no, invoice_no, age, sales_extended,age_lt_45,
age_45_120, age_gt_120)

## aging by customer

  customer_age_ar <-
  age_ar %>%
  group_by(customer_no) %>%
  summarize(total = sum(sales_extended),
            lt_45 = sum(age_lt_45),
            between_45_120 = sum(age_45_120),
            gt_120 = sum(age_gt_120)
  ) %>%
  select(customer_no, total,lt_45, between_45_120, gt_120)


knitr::kable(customer_age_ar, caption = "Customer A/R Aging") %>%
  kable_styling(bootstrap_options = "striped")


## aging of all A/R

  age_ar <-
  age_ar %>%
  summarize(total = sum(sales_extended),
            lt_45 = sum(age_lt_45),
            between_45_120 = sum(age_45_120),
            gt_120 = sum(age_gt_120)
  ) %>%
  select(total,lt_45, between_45_120, gt_120)

knitr::kable(age_ar, caption = "A/R Aging") %>%
  kable_styling(bootstrap_options = "striped")
```

Employee expenditures

Expense file is not related to anything else. I make this interesting for the fraud testing (Benford tests).

```
############################################################################
##
## 10 - Employee expenditures
############################################################################
##



no_exp_records <- floor(no_of_sales/3)
exp_string <- seq(1, no_exp_records)
exp_string <- sprintf("E%05d", exp_string)

exp_date <- rdate(no_exp_records, min = fyear_begin, max =  fyear_end)

## create some odd behavior for the Benford test

exp_set_1 <- runif(no_exp_records/3, 1, 1000)
exp_set_2 <- rpois(no_exp_records/3, 490)
exp_set_3 <- abs(rnorm(no_exp_records/3,900,200))

no_emp_records <- floor(no_exp_records/20)
emp_string <- seq(1, no_emp_records)
emp_string <- sprintf("Emp%04d", emp_string)

emp_string <- sample(
  emp_string,
  no_exp_records,
  replace = T,
  prob=runif(no_emp_records,0,1))

expenditures <- data.frame("exp_no"=exp_string ,
                      "employee_no"=emp_string ,
                      "date"= exp_date,
                      "amount"= c(exp_set_1,exp_set_2,exp_set_3)
                      )
```

Omissions, Duplications and Monetary Errors in Transactions

Test for omissions and duplication of transactions are standard in the interim tests of internal controls. The prior generation of "real world" AR and Inventory records is complete, but in this section, we remove and duplicate transactions to simulate a failure of controls in the clients accounting systems.

```
########################################################################
## 11 - Omitted, Duplicate and In-Error Records
## (errors are called "tainings"" in the monetary unit sampling vernacular)
########################################################################


# set.seed(123456)
library(tidyverse)

## create the real world transaction journals

credit_sales_journal <-
  sales_journal[,1:15] %>%
  filter(cash_not_ar == 0) %>%
  select(customer_no,
         invoice_no,
         invoice_date,
         sku,
         sales_count,
         sales_return,
         unit_cost,
         sales_unit_price,
         collection_no,
         collection_date,
         collection_amount,
         shipper_no,
         shipper_date
         )

real_world_credit_sales <- credit_sales_journal


## omitted  records

omit_row <- as.data.frame(rbinom(nrow(real_world_credit_sales),1,.05))
colnames(omit_row) = "omitted"

test <- cbind(real_world_credit_sales,omit_row)
test1 <- split(test,omit_row)
real_world_credit_sales_new <- test1$`0`

n <- nrow(test1$`1`)
cat("\n\n # omitted credit sales = ",n )
```

```
## duplicated records

dup_row <- as.data.frame(rbinom(nrow(real_world_credit_sales_new),1,.05))
colnames(dup_row) = "duplicated"


test <- cbind(real_world_credit_sales_new,dup_row)
test1 <- split(test,dup_row)
real_world_credit_sales_new <- rbind(test, test1$`1`)
real_world_credit_sales_new$dup_row = NULL

n <- nrow(test1$`1`)
cat("\n\n # duplicated credit sales = ",n )

## 'taint' 5% the values in monetary fields

real_world_credit_sales_new$taint_ed <-
    rbinom(nrow(real_world_credit_sales_new),1,.05)  ## 5%


real_world_credit_sales_new$taint_ing <-
  (real_world_credit_sales_new$sales_count /
mean(real_world_credit_sales_new$sales_count)) *
    abs(rnorm(nrow(real_world_credit_sales_new),.5, .5))

n <- nrow(real_world_credit_sales_new)
cat("\n\n # omitted credit sales = ",n )

## Cash sales - Note that these share the invoice number sequence with
credit sales

cash_sales_journal <-
  sales_journal[,1:15] %>%
  filter(cash_not_ar == 1) %>%
  select(customer_no,
         invoice_no,
         invoice_date,
         sku,
         sales_count,
         sales_return,
         unit_cost,
         sales_unit_price,
         collection_no,
         collection_date,
         collection_amount,
         shipper_no,
         shipper_date
         )

real_world_cash_sales <- cash_sales_journal
```

```
## omitted  records


omit_row <- as.data.frame(rbinom(nrow(real_world_cash_sales),1,.05))
colnames(omit_row) = "omitted"

test <- cbind(real_world_cash_sales,omit_row)
test1 <- split(test,omit_row)
real_world_cash_sales_new <- test1$`0`

n <- nrow(test1$`1`)
cat("\n\n # omitted cash sales = ",n )


## duplicated records

dup_row <- as.data.frame(rbinom(nrow(real_world_cash_sales_new),1,.05))
colnames(dup_row) = "duplicated"

test <- cbind(real_world_cash_sales_new,dup_row)
test1 <- split(test,dup_row)
n <- nrow(test1$`1`)
cat("\n\n # duplicated cash sales = ",n )

real_world_cash_sales_new <- rbind(test, test1$`1`)

n <- nrow(test1$`1`)
cat("\n\n # duplicated cash sales = ",n )

## 'taint' 5% the values in monetary fields

real_world_cash_sales_new$taint_ed <-
    rbinom(nrow(real_world_cash_sales_new),1,.05)  ## 5%

real_world_cash_sales_new$taint_ing <-
  (real_world_cash_sales_new$sales_count /
mean(real_world_cash_sales_new$sales_count)) *
    abs(rnorm(nrow(real_world_cash_sales_new),.5, .5))

n <- nrow(real_world_cash_sales_new)
cat("\n\n # tainted cash sales = ",n, "\n\n" )


## collections

real_world_collections <- collections

## omitted  records

omit_row <- rbinom(nrow(real_world_collections),1,.05)

test <- cbind(real_world_collections,omit_row)
test1 <- split(test,omit_row, drop=F)
real_world_collections_new <- test1$`0`
```

```
n <- nrow(test1$`1`)
cat("\n\n # omitted collections = ",n )


## duplicated records

dup_row <- rbinom(nrow(real_world_collections_new),1,.05)

test <- cbind(real_world_collections_new,dup_row)
test1 <- split(test,dup_row, drop=F)
real_world_collections_new <- rbind(test, test1$`1`)
real_world_collections_new$dup_row = NULL

n <- nrow(test1$`1`)
cat("\n\n # duplicated collections = ",n )

## 'taint' 5% the values in monetary fields

real_world_collections_new$taint_ed <-
    rbinom(nrow(real_world_collections_new),1,.05)  ## 5%

real_world_collections_new$taint_ing <-
    rexp(nrow(real_world_collections_new),.5)-.5   ## exponential -50%-inf%
taint

split_cash <-
  split(real_world_collections_new,
        real_world_collections_new$taint_ed)

split_cash$`1`$sales_extended <-
  split_cash$`1`$sales_extended * split_cash$`1`$taint_ing

collections <-rbind(split_cash$`1`,split_cash$`0`)

n <- nrow(split_cash$`1`)
cat("\n\n # tainted collections = ",n, "\n\n" )

## AR

## Sales-inv for the AR records


real_world_sales_journal <- sales_journal


## omitted  records

omit_row <- rbinom(nrow(real_world_sales_journal),1,.05)

test <- cbind(real_world_sales_journal,omit_row)
test1 <- split(test,omit_row, drop=F)
real_world_sales_journal_new <- test1$`0`
```

```
n <- nrow(test1$`1`)
cat("\n\n # omitted s_i   = ",n )


## duplicated records

dup_row <- rbinom(nrow(real_world_sales_journal_new),1,.05)

test <- cbind(real_world_sales_journal_new,dup_row)
test1 <- split(test,dup_row, drop=F)
real_world_sales_journal_new <- rbind(test, test1$`1`)
real_world_sales_journal_new$dup_row = NULL

n <- nrow(test1$`1`)
cat("\n\n # duplicated s_i   = ",n )

## 'taint' 5% the values in monetary fields

real_world_sales_journal_new$taint_ed <-
    rbinom(nrow(real_world_sales_journal_new),1,.05)  ## 5%

real_world_sales_journal_new$taint_ing <-
    rexp(nrow(real_world_sales_journal_new),.5)-.5   ## exponential -50%-
inf% taint

split_cash <-
  split(real_world_sales_journal_new,
        real_world_sales_journal_new$taint_ed)

split_cash$`1`$sales_extended <-
  split_cash$`1`$sales_extended * split_cash$`1`$taint_ing

sales_journal <-rbind(split_cash$`1`,split_cash$`0`)

n <- nrow(split_cash$`1`)
cat("\n\n # tainted   s_i = ",n, "\n\n" )

sales_journal <- real_world_sales_journal_new




## Inventory
## generate omitted inv'y records

real_world_po_journal <- purchase_journal

## omitted  records

omit_row <- rbinom(nrow(real_world_po_journal),1,.05)

test <- cbind(real_world_po_journal,omit_row)
test1 <- split(test,omit_row, drop=F)
real_world_po_journal_new <- test1$`0`
```

```
n <- nrow(test1$`1`)
cat("\n\n # omitted  po_inv = ",n )


## duplicated records

dup_row <- rbinom(nrow(real_world_po_journal_new),1,.05)

test <- cbind(real_world_po_journal_new,dup_row)
test1 <- split(test,dup_row, drop=F)
real_world_po_journal_new <- rbind(test, test1$`1`)
real_world_po_journal_new$dup_row = NULL

n <- nrow(test1$`1`)
cat("\n\n # duplicated  po_inv = ",n )

purchase_journal <- real_world_po_journal_new
```

Audit Tasks: Inventory and Accounts Receivable

The two major audit tasks in the sales and procurement cycle are (1) confirmation of accounts receivable, conducted by contacting customers and third parties, and (2) count of physical inventory, which I assume is conducted through one-shot physical count conducted at fiscal year end (12-31). In both cases, a full dataset of all "real world" records is generated, with the assumption that the auditor will take a sample of the clients recorded transactions, and obtain, through confirmation or counting, these "real world" values. This will require a sampling of client records to select a subset of transactions to confirm, or a set of SKUs to count, and then left_joining the sample with these "real world" values, which simulates the audit work that was done.

```


########################################################################
##
## 12 -  Simulation of completed audit work on Physical Inventory Counts
##
########################################################################


library(tidyverse)
library(lubridate)

perpetual_inventory_ledger$date <- as_date(perpetual_inventory_ledger$date)

working_perpetual <-     ## get the unit_cost from the purchase journal
  left_join(perpetual_inventory_ledger, purchase_journal[,1:5],
by=c("sku")) %>%
  group_by(sku) %>%
  arrange(date) %>%
  filter(date<=fyear_end)
```

```
real_world_ye_inventory  <-
year_end_inventory <-
  working_perpetual %>%
  as.data.frame() %>%
  filter(date<=fyear_end)%>%
  group_by(sku) %>%
  arrange(date) %>%
  slice(n()) %>%
  ungroup %>%
  select(last_transaction_date=date, sku, ye_stock_on_hand=po_count,
unit_cost)

real_world_ye_inventory$count_exception  <-
  real_world_ye_inventory$exception <-
  real_world_ye_inventory$count_exception  <-
  rbinom(nrow(real_world_ye_inventory),1,.1)  # 10% error rate

exception <- c("Obsolete_markdown" ,
               "Damaged",
               "Misclassified",
               "Refurbished_discount",
               "Open_box")

real_world_ye_inventory$actual_unit_market <-
real_world_ye_inventory$unit_cost
real_world_ye_inventory$exception <- "No exception, count is accurate"

for(i in 1:nrow(real_world_ye_inventory)){

if(real_world_ye_inventory$count_exception[i] == 1) {

real_world_ye_inventory$actual_unit_market[i] <-
  real_world_ye_inventory$unit_cost[i] *
  (1 - runif(1,0,1))

real_world_ye_inventory$exception[i] <- sample(exception,
                                        size = 1,
                                        replace=T)
}
}

real_world_ye_inventory <- as.data.frame(real_world_ye_inventory)
```

Accounts receivable confirmations

```
##########################################################################
## 13 -  Simulation of completed audit work on
##  Accounts Receivable Confirmations
########################################################################

library(tidyverse)

library(tidyverse)
library(lubridate)


fyear_end_ar_ledger <-
  sales_journal %>%
  filter(cash_not_ar == 0) %>%
  filter(collection_date > fyear_end) %>%
  select(customer_no, invoice_no, amount = sales_extended,shipper_no,
shipper_date)



neg_confirms <-
  c("No Response",
               "Unable to confirm balance",
               "Customer will not pay shipping" ,
               "Customer rejects charge",
               "Customer claims to have returned the goods",
               "Customer cannot pay",
               "Customer claims fraction of the amount charged")


real_world_fyear_end_ar_ledger <- fyear_end_ar_ledger
real_world_fyear_end_ar_ledger$confirm_exception <-
rbinom(nrow(fyear_end_ar_ledger),1,.1)  # 10% error rate
real_world_fyear_end_ar_ledger$confirm_response <-  "Confirmed, Balance OK"
real_world_fyear_end_ar_ledger$confirm_pct <-  1

for(i in 1:nrow(fyear_end_ar_ledger)){

if(real_world_fyear_end_ar_ledger$confirm_exception[i] == 1) {

real_world_fyear_end_ar_ledger$confirm_response[i] <-
    sample(neg_confirms, size = 1, replace=T)

real_world_fyear_end_ar_ledger$confirm_pct[i] <-
  runif(1,.5,.99)
}
}

real_world_fyear_end_ar_ledger <-
as.data.frame(real_world_fyear_end_ar_ledger)
```

Accounting Files for Audit

The prior code generated a set of files that contain a complete set of records of transactions, triggers and subsequent events for our simulated sales and procurement cycle. These do not appear in this form in the real world. Rather the accounting statements that auditors will find at the client's site will be organized differently. The following code reorganized the data we generated previously into files that would be typical of those found in an audit. These are written to .csv files.

```
###########################################################################
########
##
## 14 – separate po_inv and sales_journal files into component ledgers and
journals
##      write these into CSV files in the 'audit_files' directory
##
##########################################################

library(tidyverse)

## Set the directory for the new files (modify the path as needed)
default_dir <- "/home/westland/audit_analytics_book/audit_simulated_files/"

if (file.exists(default_dir)){
    setwd(default_dir)
} else {
    dir.create(default_dir)
    setwd(default_dir)
}

# sales

write.csv(real_world_cash_sales, "real_world_cash_sales.csv")
write.csv(real_world_credit_sales, "real_world_credit_sales.csv")
write.csv(sales_journal[,1:15],"sales_journal.csv")

write.csv(real_world_collections[,1:9], "real_world_collections.csv")
write.csv(collections[,1:9], "collections_journal.csv")
write.csv(deposit_daily, "deposit_daily.csv")

write.csv(fyear_end_ar_ledger,"fyear_end_ar_ledger.csv")
write.csv(real_world_fyear_end_ar_ledger,"real_world_fyear_end_ar_ledger.cs
v")
write.csv(daily_ar_balance,"daily_ar_balance.csv")


cr_limit %>%
  as.data.frame() %>%
write.csv("customer_credit_limits.csv")

write.csv(shipments_journal, "shipments_journal.csv")

# expenses
```

```
write.csv(expenditures, "expenditures.csv")

# inventory

write.csv(fyear_begin_inventory_ledger[,1:4],
"fyear_begin_inventory_ledger.csv")
write.csv(real_world_ye_inventory, "real_world_ye_inventory.csv")
write.csv(perpetual_inventory_ledger, "perpetual_inventory_ledger")

# purchases

write.csv(purchase_journal[,1:5], "purchase_journal.csv")
write.csv(ap_ledger, "ap_ledger.csv")
write.csv(receiver_journal, "receiver_journal.csv")
write.csv(disbursement_journal, "disbursement_journal.csv")



############################################################################
########
##
##    END OF PROGRAM CODE
##
############################################################################
########
```