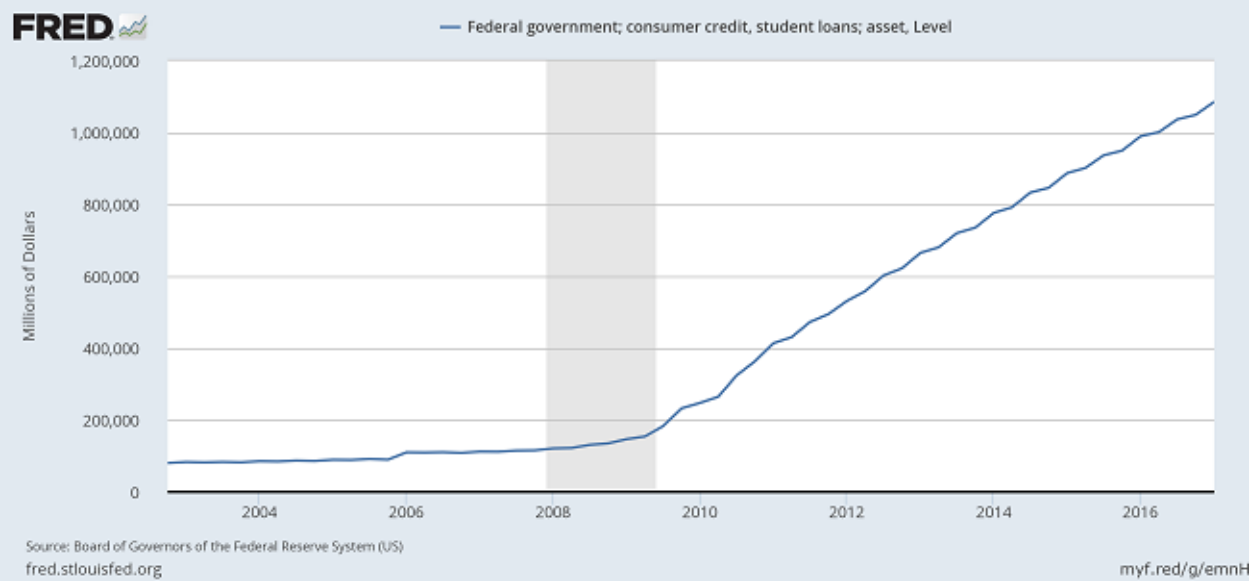
Control, Writing Functions, and Looping in R

For this lecture, we turn to a case study of student loans. We'll be using R to analyze student loan debt for an individual, in particular looking at how interest affects debt over a long term, and how different rates of repayment can affect the total amount repaid. As the end of the lecture, you will know how to:

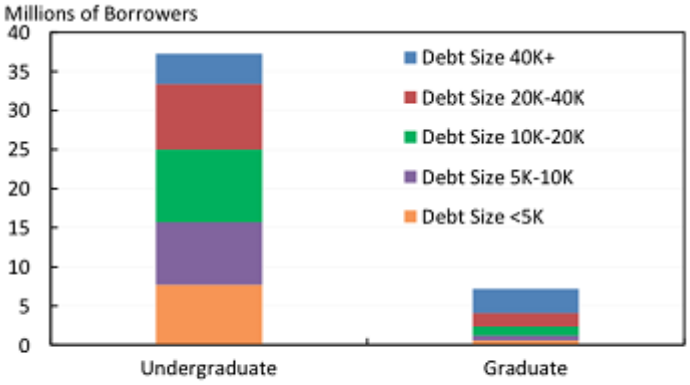
1. Use if/else statement to control the flow of your code
2. Use for loops to automate repetitive tasks and increase the reliabilty of your code
3. Create your own functions to improve code performance

This lecture will cover materials included in Chapter 19 - Functions, and Chapter 21 - Iteration

Student Loan Debt in the United States

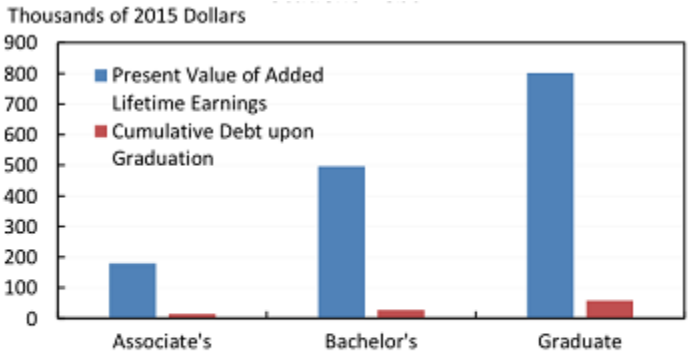


On the macroeconomic level, we can see that, particularly after the Great Recession, the amount of Federal student debt in the U.S. ballooned. In 2016 overall, Americans owed over $1 trillion in student loan debt, more than even credit card debt. Student loan debt is now the single largest bucket of consumer credit outstanding in the country. Because of the laws governing bankruptcy, student loan debt is also among the hardest debt to discharge. If a borrower can't find well-paying employment, student loan debt can hang over their head for decades.



Note: Data are as of June 2015. Undergraduate and graduate are broken apart at the loan level.
Source: Department of Education

On the microeconomic level, most undergrads who have debt end up owing between $10k and $40K. The average graduate of the class of 2016 left school holding $37,172 in student loan debt. A quarter of graduate students will leave school with more than $100,000 in debt, and 1 in 10 will owe more than $150,000. Economic research and surveys of young adults has suggested that holding this level of debt is causing many young adults to delay purchasing houses, getting married, and having children[1].



Note: Lifetime earnings are calculated by summing median annual earnings for full-time, full-year workers at every age between 25 and 64 by educational attainment, subtracting earnings for the same type of worker with only a high school degree, and converting to present value using a 3.76% discount rate.
Source: CPS ASEC 2014 and 2015; NPSAS 2012

All that being said, the returns on higher education are so great that it still makes sense for individuals to get their college degree, even if it saddles them with extra debt. The addtional earnings granted to the average person by a college degree completely dwarf the debt that person has to take on to get the degree.

Since debt is something that affect most students on an individual level, today we'll be using R to explore an individual's student debt.

Individidual Student Loans

We'll start with the data for a single student.

Read in the csv named "loan1.csv", and take a look at the contents. Since individuals' loan data is confidential, this file contains total student loan data for a hypothetical recent graduate.

```
loanData <- read.csv("../data/loan1.csv", stringsAsFactors = FALSE)

loanData
```

[1]See: "Student debt delays spending, saving - and marriage" cnn.com/2013/05/09/pf/college/student-loan-debt/

```
##    loan_id     date principal rate subsidized
## 1   51528A 8/1/2012      2000  5.8         NO
## 2   75631B 8/1/2012      1200  2.2        YES
## 3   45781A 1/1/2013      1700  4.0         NO
## 4   89634C 1/1/2013      2100  5.8         NO
## 5   12456D 1/1/2013      1500  2.0        YES
## 6   85469C 8/1/2013      2500  2.0         NO
## 7   96583B 8/1/2013      1700  2.3         NO
## 8   17536B 1/1/2014      4500  6.8         NO
## 9   32568B 1/1/2014       500  2.8        YES
## 10  32415B 1/1/2014       500  2.8        YES
## 11  78632B 8/1/2014      1000  5.0         NO
## 12  42153C 8/1/2014      1200  5.0         NO
## 13  13498R 8/1/2014      2000  2.2        YES
## 14  85974R 1/1/2015      3500  6.5         NO
## 15  16396A 1/1/2015      1500  3.5         NO
## 16  13586R 8/1/2015      3000  2.2         NO
## 17  13968B 8/1/2015      1200  5.3         NO
## 18  98563A 1/1/2016      1200  5.3        YES
## 19  68326C 1/1/2016      1500  3.5         NO
## 20  48901B 1/1/2016      2000  3.5         NO
```

Here's what the data represents:

- loan_id is an internal column to the lender, not really useful to us.
- date is the date the loan was issued to the borrower
- principal is the amount of the loan
- rate is the yearly interest rate for each loan
- subsidized denotes whether or not the loan was subsidized or unsubsidized.

A brief primer on student loans:

- Each of these twenty rows is an individual loan taken out by this one borrower. To get through college, this indivdual took out twenty loans of varying amounts, not one large loan that was cumulatively added to as the student progressed through college.
- Repayment on student loans is deferred while an individual is enrolled in higher education at least half-time, and for six months after leaving higher education. The six month period is also known as the "grace period". After this grace period ends, the graduate must start paying their loans, enter an alternative payment plan, or risk default and collections.
- If the student returns to school, either for graduate school or more undergraduate work, the loans go back into deferrment.
- Subsidized loans do not accrue interest during the grace period. Unsubsidized loans do.
- Look at the interest rates on the loans for $4,500 and $3,500. At 6.8 and 6.5%, these are the two highest interest rates the individual is paying, and they're on the highest value loans. Note also that these loans are not subsidized. When a loan is subsidized, it's not that interest doesn't get applied, it's that the Department of Education is *paying off* the interest as it accrues. The Dept. of Ed has to allocate its money carefully, so it is less likely to subsidize the biggest loans, or those with the highest interest rates.
- Under the standard repayment plan, the borrower will pay off the entirety of their student loan debt 10 years after they leave school.

Looking at the data for coding purposes:

```
str(loanData)
```

```
## 'data.frame':    20 obs. of  5 variables:
##  $ loan_id   : chr  "51528A" "75631B" "45781A" "89634C" ...
##  $ date      : chr  "8/1/2012" "8/1/2012" "1/1/2013" "1/1/2013" ...
##  $ principal : int  2000 1200 1700 2100 1500 2500 1700 4500 500 500 ...
##  $ rate      : num  5.8 2.2 4 5.8 2 2 2.3 6.8 2.8 2.8 ...
##  $ subsidized: chr  "NO" "YES" "NO" "NO" ...
```

Most of the data looks easy enough to deal with. Note that "subsidized" isn't a boolean type, but a series of "YES" and "NO". The "date" column does need to be converted to an R date-type, though.

```
loanData$date <- as.Date(loanData$date, format = "%m/%d/%Y")
```

Take a look at the second most important statistic for the borrower, the total debt:

```
select(loanData, principal) %>% sum()
```

```
## [1] 36300
```

We have a borrower slightly below average, with "just" $36,300 in debt. This is the second most important statistic because the borrower has interest due on these loans. Student loans assume a 10-year repayment plan, if the borrower only pays the monthly minimum. It is more relevant to the borrower to know the total amount of money they'll be repaying over the life of their loan. The problem we have is that, in order to calculate the total debt, we'll need to be able to tell R to only apply interest during the grace period to the unsubsidized loans. This can be done using what we call "control" statements, if and else.

If and Else

In R, and many coding languages, you can direct that code is only executed IF a certain condition is met. As a basic example:

```
x <- 35
if (x > 0) {
    print("This number is positive")
}
```

```
## [1] "This number is positive"
```

Since the condition is TRUE, the code inside the braces is executed. If the condition is FALSE, you can direct R to execute separate code using the ELSE statement:

```
x <- -2
if (x >= 0) {
    sqrt(x)
} else {
    print("Cannot compute the square root of a negative number")
}
```

```
## [1] "Cannot compute the square root of a negative number"
```

This simple bit of code uses an if/else statment to make sure that it doesn't try to take the square root of a negative. R can nest multiple IF and ELSE statements for more complicated functions.

```
grade <- 85
if (grade >= 90) {
    print("A")
} else if (grade >= 80) {
    print("B")
} else if (grade >= 70) {
    print("C")
} else if (grade >= 60) {
    print("D")
} else {
    print("F")
}
```

```
## [1] "B"
```

This code takes a numeric grade in and returns the corresponding letter grade. Note that we did not need to check to make sure that a grade is between 80 and 90 before returning a "B", we only checked that it was greater than 80. This is because the only time that the code gets to that level of checking is if it failed the first check. In other words, if a grade was greater than 90, this code never would have checked to see if it should return a B at all. Also make a note of the formatting. The "else" and "else if" statements must be on the same line as the braces for R to understand they are meant to be chained together.

Also note the very last "else" statement. It doesn't have a condition. This means that any number that fails the above checks will automatically be assigned an F. If someone accidently enters a negative value, which makes no sense, this code won't point out the error, it will just return an F. You need to be very careful with catchall statements and understand all the cases they are going to catch when writing code like this.

In Class Exercise:

Using if/else statements, build a rock/paper/scissors game.

```
player1 <- "paper"
player2 <- "scissors"

if ((player1 == "rock" & player2 == "scissors") | (player1 ==
    "paper" & player2 == "rock") | (player1 == "scissors" & player2 ==
    "paper")) {
    print("Player 1 wins!")
} else {
    print("Player 2 wins!")
}
```

```
## [1] "Player 2 wins!"
```

Calculating total debt

We can now turn to total student debt. Before we can look at repaying the loans, we need to calculate how much interest the student accrued while finishing school. We'll assume the "worst case" scencario, where the borrower only pays the minimum amount possible, only when they have to. Assuming the student graduated

on June 17, 2016, the grace period would end December 27, 2016. We'll also assume that the student enters the standard repayment plan, and that they never go back to school or have repayment deferred for any other reason.

With all this in mind, we can use if/else statments to calculate the total amount owed, based off whether or not the loan is subsidized.

As subsidized loans don't accrue interest, their value is just the value of the loan. The equation we'll need to use for unsubsidized loans is:

$$A = P(1 + rt)$$

- A is the total principal, interest included
- P is the initial principal
- r is the interest rate
- t is the time fame.

Note that you need r and t to have the same units. You can't use annual interest to directly compute monthly principal.

Looking at our data frame, we have P in the principal column. We have r, as the rate, which we'll need to adjust from annual to monthly, and from percent to decimal. We have the date the loan originated, so we just need to calculate the months between the origination date and the date the grace period ended to calcuate the time frame.

First, let's establish when the grace period ends, so we can then calculate the total amount of principal for each loan when the repayment period begins. Then, we'll initialize a new column that will represent the total principal once the repayment period ends. Because we'll also need to multiply the principal by the monthly interest rate, not the yearly, we'll convert the annual rate into a monthly decimal value. Finally, we'll initalize a column that will contain the number of months that have passed since the loan originated.

```
gracePeriodEnds <- as.Date("2016-12-15")

loanData <- loanData %>% mutate(rate = rate/12/100, currentPrincipal = 0,
    age = 0)
```

Now, to calculate the principal for unsubsidized loans, we'll need need to know the number of months that the loan has been accruing interest. This is more complicated than it seems, because months have differing lengths. 356 days does not get evenly divided into 12 months, and loan payments are based on calendar months, not theoretical ones. We can't just use division to convert years to months.

(Note: Technically, interest doesn't accure monthly, but daily. The daily interest just accumulates, then once a month it "vests", and is added to the principal. We're going with monthly interest to simplify the calculations and code.)

In R, we can use the seq() command to get around this restriction. When using seq() to build a vector between two dates, the user can specify a time period, including "month". If we make a sequence of each month between the originiation date and the date the grace period ends, we'll have a vector whose length is the number of months that have passed. Then, once we have the time since the loan originated, we can use an if statement to apply interest to the principal for the unsubsidized loans.

```
loanData[1, "age"] <- length(seq(loanData[1, "date"], gracePeriodEnds,
    by = "month"))

if (loanData[1, "subsidized"] == "NO") {
```

```
    principal <- loanData[1, "principal"]
    rate <- loanData[1, "rate"]
    age <- loanData[1, "age"]

    loanData[1, "currentPrincipal"] <- principal * (1 + rate *
        age)

} else {

    loanData[1, "currentPrincipal"] <- loanData[1, "principal"]
}

loanData[1, ]
```

```
##   loan_id       date principal        rate subsidized currentPrincipal age
## 1  51528A 2012-08-01      2000 0.004833333         NO         2512.333  53
```

Checking the data, we can see that in just the time it took for this student to graduate, they accrued an extra $512 of debt. They took out a loan for $2,000, but they have to pay off a loan of $2,512.

Our code worked on the first loan, so now we repeat the process for the next row, and the row after. To do this, we can just copy and paste the code we've already used, but change it to address the second row, and then the third, instead of the first.

```
# Row 2
loanData[2, "age"] <- length(seq(loanData[2, "date"], gracePeriodEnds,
    by = "month"))

if (loanData[2, "subsidized"] == "NO") {

    principal <- loanData[2, "principal"]
    rate <- loanData[2, "rate"]
    age <- loanData[2, "age"]

    loanData[2, "currentPrincipal"] <- principal * (1 + rate *
        age)

} else {

    loanData[2, "currentPrincipal"] <- loanData[2, "principal"]
}

# Row 3
loanData[3, "age"] <- length(seq(loanData[3, "date"], gracePeriodEnds,
    by = "month"))

if (loanData[3, "subsidized"] == "NO") {

    principal <- loanData[3, "principal"]
    rate <- loanData[3, "rate"]
    age <- loanData[3, "age"]

    loanData[3, "currentPrincipal"] <- principal * (1 + rate *
```

```
        age)

} else {

    loanData[3, "currentPrincipal"] <- loanData[3, "principal"]
}

head(loanData[, c("date", "principal", "rate", "subsidized",
    "currentPrincipal")], 5)
```

```
##         date principal        rate subsidized currentPrincipal
## 1 2012-08-01      2000 0.004833333         NO         2512.333
## 2 2012-08-01      1200 0.001833333        YES         1200.000
## 3 2013-01-01      1700 0.003333333         NO         1972.000
## 4 2013-01-01      2100 0.004833333         NO            0.000
## 5 2013-01-01      1500 0.001666667        YES            0.000
```

After three iterations of our code, we have just the first three loans assessed. The second loan was subsidized, which mean that the currentPrincipal is just the principal, as expected. The third loan accrued an extra $272 of principal while this student was in college. On just the first two unsubsidized loans, the borrower had added $784 of extra debt.

Looking at our code, note that editing the code to run on each one of these rows requires NINE changes. If you miss even one, the code will not work. It may just break, or it may instead use the wrong rate or principal to do the calculations. These sorts of bugs are very hard for the user to catch, since it looks like the code is returning valid output. With this data, we can't use the filter() function because we'll need the entire data frame to calculate how interest payments are affecting this student. Filter() will make us break this into two data frames and complicate our code. We're forced to work row-by-row.

Remember that one of the reasons we prefer to code over other forms of data analysis is that coding allows us to make the computer handle repetitive tasks. One way to let the computer do this is through a "for loop".

For Loops

In R, and many other coding languages, programmers can create loops to automate repetitive tasks. Each loops needs three parts. An *index*, or a way to note the values you want to run the code over, a *variable* (often i), that lets the computer adjust the code on the fly, and the *code* that needs to be exectued. For loops take the form:

for(*variable* in *index*) {
Code to be exectuted
}

The index is simply a vector that R uses to note what values to run the code over. With data frames, you'll often want to run the code over every row or every column. In these cases, the index is just a list of all rows or columns. In other cases, you may want to pick specific columns, in which case you'd simply feed in a vector of the selected few.

Note the language inside the parantheses. for i in *index*. What this tells R is to, one by one, assign each value found in the index to i, and then run the code in the braces with that i in turn. Let's look at a simple example to help understand how this works.

```
boringVector <- c(3, 5, 9, 11, 16, 21, 44)
```

```
for (i in c(1, 2, 3, 4, 5, 6, 7)) {
    print(boringVector[i]^2)
}
```

```
## [1] 9
## [1] 25
## [1] 81
## [1] 121
## [1] 256
## [1] 441
## [1] 1936
```

For this simple loops, R takes each entry in the boring vector and squares it. The index is just the numbers 1 through 7. We could also use the ind

For each of these numbers, in turn, R prints the square of that number. Equivalent code could be written as:

```
i <- 1
boringVector[i]^2
```

```
## [1] 9
```

```
i <- 2
boringVector[i]^2
```

```
## [1] 25
```

```
i <- 3
boringVector[i]^2
```

```
## [1] 81
```

```
i <- 4
boringVector[i]^2
```

```
## [1] 121
```

```
i <- 5
boringVector[i]^2
```

```
## [1] 256
```

```
i <- 6
boringVector[i]^2
```

```
## [1] 441
```

```
i <- 7
boringVector[i]^2
```

```
## [1] 1936
```

Note that the code executed is the same in each instance, always i ^ 2. By using a loop, we save ourselves the trouble of copy/paste/editing the code over and over.

We can further simplify this code by using the seq() command or the : operator to generate the index instead of typing it out. Remember that seq() and : both return a vector, and a vector is what we use for indexing in R.

```
for (i in seq(1, 7)) {
    print(boringVector[i]^2)
}
```

```
## [1] 9
## [1] 25
## [1] 81
## [1] 121
## [1] 256
## [1] 441
## [1] 1936
```

```
for (i in 1:7) {
    print(boringVector[i]^2)
}
```

```
## [1] 9
## [1] 25
## [1] 81
## [1] 121
## [1] 256
## [1] 441
## [1] 1936
```

In Class Exercise:

Use a for loop to generate the first 15 digits of the Fibonacci Sequence

```
fibSeq <- c(0, 1)

for (i in 3:15) {
    fibNext <- fibSeq[i - 1] + fibSeq[i - 2]

    fibSeq <- c(fibSeq, fibNext)
}

fibSeq
```

```
##  [1]   0   1   1   2   3   5   8  13  21  34  55  89 144 233 377
```

Converting the Princpal Code into a For Loop

Let's convert the code that determines starting principal into a for loop to save us a lot of effort.

First, we need the index. We want to iterate over every row in the data frame. Using the [] notation lets us just use the number index of each row. Our index is the first row through the last. We can use nrow() to have R find the last row of the data frame for us.

```
1:nrow(loanData)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

We'll just use i again, for simplicity's sake. With the index and the variable set, we just need to edit our code to use i instead of the row number. Everywhere we had to change the index from 1, to 2, to 3 up above, we just replace with i. This will tell R to use the ith row, instead of a specific one, allowing R to iterate over each i in our index, which will cover the whole data frame.

```
for (i in 1:nrow(loanData)) {
    loanData[i, "age"] <- length(seq(loanData[i, "date"], gracePeriodEnds,
        by = "month"))

    if (loanData[i, "subsidized"] == "NO") {

        principal <- loanData[i, "principal"]
        rate <- loanData[i, "rate"]
        age <- loanData[i, "age"]

        loanData[i, "currentPrincipal"] <- principal * (1 + rate *
            age)

    } else {

        loanData[i, "currentPrincipal"] <- loanData[i, "principal"]
    }
}

select(loanData, date, principal, rate, subsidized, currentPrincipal)
```

```
##          date principal        rate subsidized currentPrincipal
## 1  2012-08-01      2000 0.004833333         NO         2512.333
## 2  2012-08-01      1200 0.001833333        YES         1200.000
## 3  2013-01-01      1700 0.003333333         NO         1972.000
## 4  2013-01-01      2100 0.004833333         NO         2587.200
## 5  2013-01-01      1500 0.001666667        YES         1500.000
## 6  2013-08-01      2500 0.001666667         NO         2670.833
## 7  2013-08-01      1700 0.001916667         NO         1833.592
## 8  2014-01-01      4500 0.005666667         NO         5418.000
## 9  2014-01-01       500 0.002333333        YES          500.000
## 10 2014-01-01       500 0.002333333        YES          500.000
## 11 2014-08-01      1000 0.004166667         NO         1120.833
## 12 2014-08-01      1200 0.004166667         NO         1345.000
## 13 2014-08-01      2000 0.001833333        YES         2000.000
## 14 2015-01-01      3500 0.005416667         NO         3955.000
```

```
## 15 2015-01-01      1500 0.002916667         NO         1605.000
## 16 2015-08-01      3000 0.001833333         NO         3093.500
## 17 2015-08-01      1200 0.004416667         NO         1290.100
## 18 2016-01-01      1200 0.004416667        YES         1200.000
## 19 2016-01-01      1500 0.002916667         NO         1552.500
## 20 2016-01-01      2000 0.002916667         NO         2070.000
```

With just one bit of code writing, we covered 20 repetitions of the same task.

Looking at the data, it's clear why subsidized loans are so much more desireable for students than unsubsidized. On just the loan for $4,500, the student accrued an extra $918 in debt before their first payment was scheduled.

```
select(loanData, currentPrincipal) %>% sum()
```

```
## [1] 39925.89
```

While the student took out "only" $36,300 in loans, when their first payment is actually due they already owe $39,925.89. This represents nearly $4,000 just in interest accrued before they actually pay a dime towards their loans.

In the above code, pay attention to the fact that *everything* inside of the for loop gets run every single time the loop iterates. We initialized all of our new columns outside of the loop because, if we initialized them inside the loop, they'd be reset to zero every time the loop interates. When working with loops, take care with initalizing variables, and where you put the initalization in the code flow.

Paying off the loan

Now that we have the initial amount owed by the student, we can start looking at how to pay off these loans.

The formula for finding the minimum payment is:

$$M = P\left(\frac{r}{1-(1+r)^{-t}}\right)$$

- M is the monthly payment, in dollars
- P is the initial principal
- r is the interest rate, in decimal
- t is the duration of the loan

Note that r and t must use the same units of time. Years or months. Loans payments rarely come due annually, so we'll again be using months as our unit of time. This means that t = 120 months for our example.

We have the initial principal for each loan in our new column, thanks to our for loop from above. Since the interest rates vary depending on the loan, we'll need to again use a for loop to calculate the minimum monthly payment. However, since we're out of the grace period, there's no need to apply different rules for subsidized/unsubsidized loans.

First, we need to initalize the column that we'll be using to store the monthly minimum payment for each individual loan. Recall that we do this outside of the loop to prevent overwriting the new column.

```
loanData <- mutate(loanData, minPayment = 0)
```

Now we'll start the for loop, again using an index that contains each row of the data frame, and telling R to iterate through each row in turn, calculating the minimum payment for each loan, and then the total minimum amount the student must pay every month.

```
for (i in 1:nrow(loanData)) {
    principal <- loanData[i, "currentPrincipal"]
    rate <- loanData[i, "rate"]

    loanData$minPayment[i] <- principal * (rate/(1 - (1 + rate)^-120))

}

select(loanData, principal, rate, subsidized, currentPrincipal,
    minPayment)
```

```
##    principal        rate subsidized currentPrincipal minPayment
## 1       2000 0.004833333         NO         2512.333  27.640392
## 2       1200 0.001833333        YES         1200.000  11.149428
## 3       1700 0.003333333         NO         1972.000  19.965541
## 4       2100 0.004833333         NO         2587.200  28.464067
## 5       1500 0.001666667        YES         1500.000  13.802018
## 6       2500 0.001666667         NO         2670.833  24.575260
## 7       1700 0.001916667         NO         1833.592  17.118997
## 8       4500 0.005666667         NO         5418.000  62.350523
## 9        500 0.002333333        YES          500.000   4.782014
## 10       500 0.002333333        YES          500.000   4.782014
## 11      1000 0.004166667         NO         1120.833  11.888176
## 12      1200 0.004166667         NO         1345.000  14.265812
## 13      2000 0.001833333        YES         2000.000  18.582380
## 14      3500 0.005416667         NO         3955.000  44.908225
## 15      1500 0.002916667         NO         1605.000  15.871182
## 16      3000 0.001833333         NO         3093.500  28.742296
## 17      1200 0.004416667         NO         1290.100  13.873470
## 18      1200 0.004416667        YES         1200.000  12.904554
## 19      1500 0.002916667         NO         1552.500  15.352031
## 20      2000 0.002916667         NO         2070.000  20.469375
```

```
select(loanData, minPayment) %>% sum()
```

```
## [1] 411.4878
```

```
select(loanData, minPayment) %>% sum() * 12 * 10
```

```
## [1] 49378.53
```

This student will owe $411.49 every month for 10 years, for a grand total of $49,378.53. This student is paying over $13,000 in interest over the life of their student loans. Nearly a quarter of the total bill the student pays is spent towards interest.

This naturally raises the question, how much money can this student save by paying more towards their student loans each month? How much will just $10 more every month pay off? How about $50, or $100?

But comparing scenarios like we would want to do would mean we need to rewrite our code every single time, catching every instance of the payment amount. This is tedious and prone to error. So instead, we're going to write our first custom function

In our previous example, we had a set duration of 10 years. In the case where we're paying more than the minimum, the duration becomes a dependent variable, and the amount paid the independent, so we'll need to develop a new formula. We want to look at the balance on the loan after each month's payment. The formula for the balance after t time periods has passed is:

$$B = P(1+r)^n - \frac{M}{r}[(1+r)^n - 1]$$

Since we're looking at every month, the number of payments between each month is 1. The formula above simplifies down to just the previous balance, plus interest, minus the monthly payment:

$$B = P(1+r) - M$$

Let's build a data frame to look at how the balance of the loan goes down over time. We care more about the duration of the loan in number of months, rather than actual dates, so let's not bother with wrangling dates.

```
loanBalance <- data.frame(month = seq(1:120), curBalance = 0)
```

With an initialized data frame, we can start thinking about how best to set up the code to calculate monthly balances. While we're thinking about just one loan right now, we should have, in the back of our minds, the fact that there's 19 other loans also to deal with, and so we should build the code to be flexible. Let's start by popping out the starting balance and interest rate from the first loan. Since the rate never changes, we don't need to track changes over time. We can just store it as a static variable and save some typing. Let's first test it by assuming the borrower is just paying the minimum, or $27.64 a month.

```
loanBalance[1, "curBalance"] <- loanData[1, "currentPrincipal"]
rate <- loanData[1, "rate"]
payment <- loanData[1, "minPayment"]
```

Since the balance is changing over time, we'll have to again iterate, which means a for loop. This time, we already have our first value. In fact we need that initial value to calculate any of the rest of the values. We only care about rows 2:120, which means that's our index. Loops can complicate code, so we're also going to focus on readability.

```
for (i in 2:nrow(loanBalance)) {
    balance <- loanBalance[i - 1, "curBalance"]

    loanBalance[i, "curBalance"] <- balance * (1 + rate) - (payment)
}

head(loanBalance, 10)
```

```
##    month curBalance
## 1      1   2512.333
## 2      2   2496.836
## 3      3   2481.264
## 4      4   2465.616
## 5      5   2449.893
## 6      6   2434.093
## 7      7   2418.218
## 8      8   2402.265
## 9      9   2386.236
## 10    10   2370.129
```

```
tail(loanBalance, 10)
```

```
##     month curBalance
## 111   111  269.19604
## 112   112  242.85676
## 113   113  216.39018
## 114   114  189.79567
## 115   115  163.07262
## 116   116  136.22042
## 117   117  109.23842
## 118   118   82.12602
## 119   119   54.88257
## 120   120   27.50744
```

Looks like the code worked. On the last month of the loans, the borrower owes $27.51, or the $27.64 minimum with a small rounding error. Now let's edit the payment to assume just $5 more a month towards this loan, and rerun our loop.

```
payment <- loanData[1, "minPayment"] + 5

for (i in 2:nrow(loanBalance)) {
    balance <- loanBalance[i - 1, "curBalance"]

    loanBalance[i, "curBalance"] <- balance * (1 + rate) - (payment)
}

head(loanBalance, 10)
```

```
##    month curBalance
## 1      1   2512.333
## 2      2   2491.836
## 3      3   2471.239
## 4      4   2450.543
## 5      5   2429.747
## 6      6   2408.851
## 7      7   2387.853
## 8      8   2366.754
## 9      9   2345.553
## 10    10   2324.249
```

```
tail(loanBalance, 10)
```

```
##     month curBalance
## 111   111  -454.5158
## 112   112  -489.3530
## 113   113  -524.3586
## 114   114  -559.5334
## 115   115  -594.8782
## 116   116  -630.3938
## 117   117  -666.0811
## 118   118  -701.9409
## 119   119  -737.9740
## 120   120  -774.1813
```

The start of the series looks good, but there's a problem at the end. The borrower is paying off their loan faster, so the duration is no longer 120 months. Negative balances don't make sense, we need to modify our code to filter them out of the data.

In Class Exercise

Use if/else statments to correct the "making payments on a negative balance" bug.

```
payment <- loanData[1, "minPayment"] + 5

for (i in 2:nrow(loanBalance)) {
    balance <- loanBalance[i - 1, "curBalance"]


    if (balance <= payment) {
        loanBalance[i, "curBalance"] <- 0
    } else {
        loanBalance[i, "curBalance"] <- balance * (1 + rate) -
            (payment)
    }
}

head(loanBalance, 10)
```

```
##    month curBalance
## 1      1   2512.333
## 2      2   2491.836
## 3      3   2471.239
## 4      4   2450.543
## 5      5   2429.747
## 6      6   2408.851
## 7      7   2387.853
## 8      8   2366.754
## 9      9   2345.553
## 10    10   2324.249
```

```
tail(loanBalance, 10)
```

```
##     month curBalance
## 111   111          0
## 112   112          0
## 113   113          0
## 114   114          0
## 115   115          0
## 116   116          0
## 117   117          0
## 118   118          0
## 119   119          0
## 120   120          0
```

User Defined Functions

There is also another problem. Since we're just copy and pasting the for loop code, we're overwriting our data frame every time we change the payment amount. We don't have a good way to compare how different

payment amounts will affect total repayment. Plus, we have 19 other loans waiting for us to analyze. We'll have to edit and copy a lot of code to handle the full problem. Here, as above, we're going to let R automate away repetitive tasks.

There are two ways to let R handle this. One is to put our current for loop inside another for loop that will iterate over all 20 loans, known as "nesting" the for loops. This is not ideal, since it isn't a very readable solution. Nesting loops like this can also make finding bugs very difficult. Instead, we are going to start writing our own functions.

You should be very used to using functions in R. You've been using functions from both the base R library, as well as functions from specialized packages, like dplyr. Now we'll start learning how to create your own functions to make your code more readable and portable.

The format for defining your own functions in R is:

functionName <- function(*arg1, arg2...*) {
code to execute
return(*value*)
}

As a simple example, we can convert some of the small code above into functions.

```
calculateRoots <- function(x) {
    if (x >= 0) {
        return(sqrt(x))
    } else {
        return("Cannot compute the square root of a negative number")
    }
}

calculateRoots(23)
```

```
## [1] 4.795832
```

```
calculateRoots(-25)
```

```
## [1] "Cannot compute the square root of a negative number"
```

```
calculateRoots(40401)
```

```
## [1] 201
```

```
calculateRoots(-40401)
```

```
## [1] "Cannot compute the square root of a negative number"
```

```
convertLetterGrade <- function(grade) {
    if (grade >= 90) {
        return("A")
    } else if (grade >= 80) {
        return("B")
    } else if (grade >= 70) {
        return("C")
```

```
    } else if (grade >= 60) {
        return("D")
    } else {
        return("F")
    }
}
convertLetterGrade(76)
```

```
## [1] "C"
```

```
convertLetterGrade(85)
```

```
## [1] "B"
```

```
convertLetterGrade(55)
```

```
## [1] "F"
```

In the above cases, all the editing we needed to do was change how the code output the results, by changing "print" to "return".

When you're naming functions, like when you name variables, you'll want to be descriptive. "doMath" isn't a good function name. "myFunction", "func1, func2, func3", and the like are also all bad. Make sure that any abbreviations you use will make sense to future users. "bal" is a good abbreviation for "balance". "B" probably won't be. Would "chr" make sense for a charge to the account? Or will it confuse users into thinking the function manipulates characters? Try to use verbs, to remind yourself and other users that this code *does* something.

In Class Exercise:

Convert the Fibonnaci code from above into a function that takes an integer, "n" in for how many digits of the sequence to compute.

```
fibonacci <- function(n = 3) {
    fibSeq <- c(0, 1)

    for (i in 3:n) {
        fibNext <- fibSeq[i - 1] + fibSeq[i - 2]

        fibSeq <- c(fibSeq, fibNext)
    }

    return(fibSeq[1:n])
}

fibonacci(10)
```

```
##  [1]  0  1  1  2  3  5  8 13 21 34
```

```
fibonacci(15)
```

```
##  [1]   0   1   1   2   3   5   8  13  21  34  55  89 144 233 377
```

```
fibonacci(20)
```

```
##  [1]    0    1    1    2    3    5    8   13   21   34   55   89  144  233
## [15]  377  610  987 1597 2584 4181
```

In Class Exercise:

Convert the code that calculates monthly balances into a function called "calculateMonthlyBal". It should take the initial balance, as well as the monthly rate and payment amount in as variables.

```
calculateMonthlyBal <- function(initialBalance, rate, payment) {
    loanBal <- data.frame(month = seq(1:120), curBal = 0)
    loanBal[1, "curBal"] <- initialBalance

    for (i in 2:nrow(loanBal)) {
        balance <- loanBal[i - 1, 2]

        if (balance <= payment) {
            loanBal[i, 2] <- 0
        } else {
            loanBal[i, 2] <- balance * (1 + rate) - (payment)
        }
    }

    return(loanBal)
}

balMinPayment <- calculateMonthlyBal(loanData[1, "currentPrincipal"],
    loanData[1, "rate"], loanData[1, "minPayment"])
head(balMinPayment)
```

```
##   month   curBal
## 1     1 2512.333
## 2     2 2496.836
## 3     3 2481.264
## 4     4 2465.616
## 5     5 2449.893
## 6     6 2434.093
```

```
tail(balMinPayment)
```

```
##     month    curBal
## 115   115 163.07262
## 116   116 136.22042
## 117   117 109.23842
## 118   118  82.12602
## 119   119  54.88257
## 120   120  27.50744
```

Working With Your Own Functions

Note that we have a function that returns a data frame. As you can see, this allows you to assign the output to a new variable, which means you can experiment with Now, instead of copy/pasting/editing code, you can just call a single function on each of the 20 loans. This makes your code much more readable, and it makes it easier to find bugs. Once you know the code to calculate monthly balances works, any bugs you find have to be outside it.

With our new function in hand, let's make a loop that will let us explore how just a few extra dollars a month will affect the total payments on the loan. Our final product will be a data frame, similar to loanBalance, but one that has the sum of all the loans. So, what we need to do is start with a balance of 0 for each month, and add, one by one, the monthly balance of each of the twenty loans.

```
totalMonthlyBal <- data.frame(month = seq(1:120), balance = 0)

for (i in 1:nrow(loanData)) {
    principal <- loanData[i, "currentPrincipal"]
    rate <- loanData[i, "rate"]
    payment <- loanData[i, "minPayment"]

    individualMonthlyBal <- calculateMonthlyBal(principal, rate,
        payment)

    totalMonthlyBal[, "balance"] <- totalMonthlyBal[, "balance"] +
        individualMonthlyBal[, "curBal"]
}

head(totalMonthlyBal)
```

```
##   month  balance
## 1     1 39925.89
## 2     2 39658.61
## 3     3 39390.39
## 4     4 39121.25
## 5     5 38851.17
## 6     6 38580.16
```

```
tail(totalMonthlyBal)
```

```
##     month   balance
## 115   115 2436.9863
## 116   116 2034.5916
## 117   117 1630.6976
## 118   118 1225.2980
## 119   119  818.3863
## 120   120  409.9558
```

With our new loop and our new function, we can calculate the monthly balance. By submitting a different payment to the argument in the calculateMonthlyBal() function call, we can also see how different payments will affect the loan. But, we're back at our old problem, where each time we run the loop, we overwrite the totalMonthlyBal data frame. We can't compare payment schemes very easily.

In Class Exercise:

Convert the code we've been working on all class into one function that takes in data like in the csv we used, the date that payments begin, and a modifier for any additional amount to increase the monthly payments, and returns a data frame simliar tototalMonthlyBal.

```
calcTotalMonthlyBal <- function(loanData, gracePeriodEnds, paymentMod = 0) {
    library(dplyr)

    # Make sure the dates are dates
    loanData$date <- as.Date(loanData$date, format = "%m/%d/%Y")
    gracePeriodEnds <- as.Date(gracePeriodEnds)

    # Now, get the data frame ready for all the calculations
    loanData <- loanData %>% mutate(rate = rate/12/100, currentPrincipal = 0,
        age = 0, minPayment = 0)

    # First, calculate the balance after the grace period ends
    for (i in 1:nrow(loanData)) {
        loanData[i, "age"] <- length(seq(loanData[i, "date"],
            gracePeriodEnds, by = "month"))


        if (loanData[i, "subsidized"] == "NO") {
            principal <- loanData[i, "principal"]
            rate <- loanData[i, "rate"]
            age <- loanData[i, "age"]

            loanData[i, "currentPrincipal"] <- principal * (1 +
                rate * age)

        } else {
            loanData[i, "currentPrincipal"] <- loanData[i, "principal"]
        }
    }

    # Now, the minimum payment
    for (i in 1:nrow(loanData)) {
        principal <- loanData[i, "currentPrincipal"]
        rate <- loanData[i, "rate"]

        loanData$minPayment[i] <- principal * (rate/(1 - (1 +
            rate)^-120))
    }

    # Now, the monthly loan balance
    totalMonthlyBal <- data.frame(month = seq(1:120), balance = 0)

    for (i in 1:nrow(loanData)) {
        principal <- loanData[i, "currentPrincipal"]
        rate <- loanData[i, "rate"]
        payment <- loanData[i, "minPayment"] + paymentMod/nrow(loanData)

        individualMonthlyBal <- calculateMonthlyBal(principal,
```

```
            rate, payment)

        totalMonthlyBal[, "balance"] <- totalMonthlyBal[, "balance"] +
            individualMonthlyBal[, "curBal"]
    }

    return(totalMonthlyBal)
}
```

In Class Exercise

Using calcTotalMonthlyBal, graph the monthly balance, for if the borrower pays just the minimum, if they pay an extra $20 each month on their loans, and if they pay an extra $100/month on their loans.

```
library(ggplot2)
```

```
## Warning: package 'ggplot2' was built under R version 3.3.3
```

```
balMinPayment <- calcTotalMonthlyBal(loanData, gracePeriodEnds = "2016-12-31")

balPlus20 <- calcTotalMonthlyBal(loanData, gracePeriodEnds = "2016-12-31",
    paymentMod = 20)
colnames(balPlus20) <- c("month", "balance20")

balPlus100 <- calcTotalMonthlyBal(loanData, gracePeriodEnds = "2016-12-31",
    paymentMod = 100)
colnames(balPlus100) <- c("month", "balance100")

plotBals <- left_join(balMinPayment, balPlus20)
```

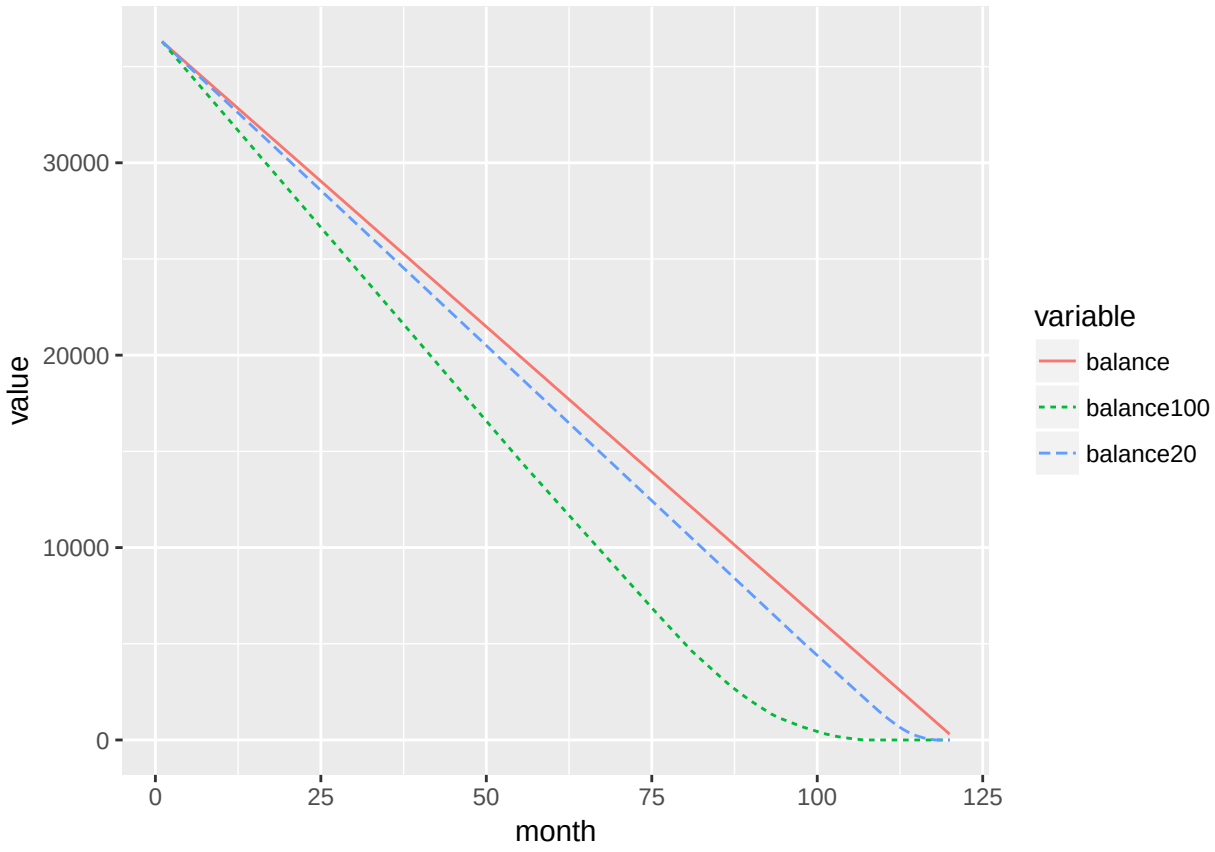
```
## Joining, by = "month"
```

```
plotBals <- left_join(plotBals, balPlus100)
```

```
## Joining, by = "month"
```

```
gatheredPlotBals <- gather(plotBals, key = "variable", value = "value",
    balance:balance100)

ggplot(gatheredPlotBals, aes(x = month, y = value, color = variable,
    linetype = variable)) + geom_line()
```


30000
20000
10000
0
value
0
25
50
75
100
125
month
variable
balance
balance100
balance20