**swirl Lesson 1: Basic Building Blocks**

Agora, vamos criar uma pequena coleção de números chamada vetor. Qualquer objeto que contém dados é chamado de estrutura de dados e vetores numéricos são os mais simples tipo de estrutura de dados em R. Na verdade, mesmo um único número é considerado um vetor de comprimento um.

A maneira mais fácil de criar um vetor é com a função c (), que significa 'concatenar' ou 'combinar'. Para criar um vetor contendo os números 1.1, 9 e 3,14, tipo c (1,1, 9, 3,14). Experimente agora e armazene o resultado em uma variável chamada z.

Vetores numéricos podem ser usados ​​em expressões aritméticas. Digite o seguinte para ver o que acontece: z \* 2 + 100.

Primeiro, R multiplicou cada um dos três elementos em z por 2. Em seguida, acrescentou 100 a cada elemento para obter o resultado que você vê acima.

Outros operadores aritméticos comuns são `+`, `-`,` / `e` ^ `(onde x ^ 2 significa 'x ao quadrado'). Para obter a raiz quadrada, use a função sqrt () e para obter o valor absoluto, use a função abs ().

Pegue a raiz quadrada de z - 1 e atribua-a a uma nova variável chamada my\_sqrt.

R primeiro subtraiu 1 de cada elemento de z, depois tirou a raiz quadrada de cada elemento. Isso deixa você com um vetor do mesmo comprimento como o vetor original z. um vetor de comprimento 3

Agora, crie uma nova variável chamada my\_div que obtém o valor de z dividido por my\_sqrt.

O primeiro elemento de my\_div é igual ao primeiro elemento de z dividido pelo primeiro elemento de my\_sqrt e assim por diante. Quando dados dois vetores do mesmo comprimento, R simplesmente executa o especificado operação aritmética (`+`, `-`,` \* `, etc.) elemento por elemento. Se os vetores forem de comprimentos diferentes, R 'recicla' o vetor mais curto até que tenha o mesmo comprimento como o vetor mais longo.

Quando fizemos z \* 2 + 100 em nosso exemplo anterior, z era um vetor de comprimento 3, mas tecnicamente 2 e 100 são vetores de comprimento 1.

Nos bastidores, R está 'reciclando' o 2 para fazer um vetor de 2s e o 100 para fazer um vetor de 100s. Em outras palavras, quando você pede a R para calcular z \* 2 + 100, o que ele realmente calcula é o seguinte: z \* c (2, 2, 2) + c (100, 100, 100).

Digite c (1, 2, 3, 4) + c (0, 10) no console para ver como R adiciona dois vetores de comprimento diferente. Não atribua o resultado a uma variável.

Se o comprimento do vetor mais curto não se divide uniformemente no comprimento do vetor mais longo, R ainda aplicará o método de 'reciclagem', mas lançará um aviso para informá-lo de que algo suspeito pode estar acontecendo. “comprimento do objeto maior não é múltiplo do comprimento do objeto menor”

Experimente c (1, 2, 3, 4) + c (0, 10, 100) como exemplo.

Mensagem de aviso:

Em c (1, 2, 3, 4) + c (0, 10, 100): “comprimento do objeto maior não é múltiplo do comprimento do objeto menor”

No início da lição, você calculou z \* 2 + 100. Vamos fingir que você cometeu um erro e que pretendia adicionar 1000 em vez de 100. Você pode digitar novamente a expressão, ou ...

Em muitos ambientes de programação, a seta para cima percorrerá os comandos anteriores. Tente pressionar a seta para cima no teclado até chegar a este comando (z \* 2 + 100), em seguida, altere 100 para 1000 e pressione Enter. Se a seta para cima não trabalhar para você, basta digitar o comando corrigido.

Finalmente, vamos fingir que você gostaria de ver o conteúdo de uma variável que você criado anteriormente, mas você não consegue se lembrar se o nomeou my\_div ou myDiv.

Você pode tentar os dois e ver o que funciona, ou ...

my\_sqrt <- sqrt (z-1) ## Raiz quadrada, será subraido 1 em cada elemento de z e depois será retirado da raiz quadrada de cada elemento

Você pode digitar as duas primeiras letras do nome da variável e, em seguida, pressionar a tecla Tab (possivelmente mais de uma vez). A maioria dos ambientes de programação fornecerá uma lista de variáveis ​​que você criou que começam com 'meu'. Isso é chamado preenchimento automático e pode ser muito útil quando você tem muitas variáveis ​​em seu área de trabalho. De uma chance. (Se o preenchimento automático não funcionar para você, basta digitar my\_div e pressione Enter.)

**2: Workspace and Files**

Nesta lição, você aprenderá como examinar seu espaço de trabalho local em R e começar para explorar a relação entre sua área de trabalho e o sistema de arquivos de sua máquina.

Como sistemas operacionais diferentes têm convenções diferentes com relação a coisas como caminhos de arquivo, as saídas desses comandos podem variar entre as máquinas.

No entanto, é importante notar que R fornece uma API comum (um conjunto comum de comandos) para interagir com arquivos, dessa forma seu código funcionará em diferentes tipos de computadores.

Determine qual diretório sua sessão R está usando como seu diretório de trabalho atual usando getwd ().

Liste todos os objetos em seu espaço de trabalho local usando ls ().

Alguns comandos R são iguais aos comandos equivalentes no Linux ou no Mac.

Os sistemas operacionais Linux e Mac são baseados em um sistema operacional chamado Unix. É sempre uma boa ideia aprender mais sobre o Unix!

Liste todos os arquivos em seu diretório de trabalho usando list.files () ou dir ().

Verifique a página de ajuda de list.files com o comando ?list.files

Uma das partes mais úteis de qualquer arquivo de ajuda do R é a seção Consulte também. Leia essa seção para list.files. Algumas dessas funções podem ser usadas em partes posteriores desta lição.

Usar a função args () em um nome de função também é uma maneira útil de ver o que argumentos que uma função pode receber.

Use a função args () para determinar os argumentos para list.files ().

args(list.files)

Atribua o valor do diretório de trabalho atual a uma variável chamada "old.dir".

old.dir <- getwd()

Usaremos old.dir no final desta lição para voltar ao lugar que

começado. Muitas funções de consulta como getwd () têm a propriedade útil de

retornar a resposta à pergunta como resultado da função.

Use dir.create () para criar um diretório no diretório de trabalho atual chamado "testdir". Digite dir.create ("testdir") para criar um diretório no diretório de trabalho atual chamado "testdir".

Faremos todo o nosso trabalho neste novo diretório e, em seguida, excluí-lo-emos depois de feito. Este é o analógico R para "Tire apenas fotos, deixe apenas pegadas."

Defina seu diretório de trabalho como "testdir" com o comando setwd ().

setwd("testdir")

Em geral, você vai querer que seu diretório de trabalho esteja em algum lugar adequado, talvez criado para o projeto específico no qual você está trabalhando. Na verdade, organizar seu trabalho em pacotes R usando RStudio é uma excelente opção. Verifique o RStudio em http://www.rstudio.com/

Crie um arquivo em seu diretório de trabalho chamado "mytest.R" usando o file.create () função.

file.create("mytest.R")

Seu sistema operacional fornecerá ferramentas mais simples para esses tipos de tarefas, mas ter a capacidade de manipular arquivos programaticamente é útil. Agora você pode tentar excluir mytest.R usando file.remove ('mytest.R'), mas isso não funcionará, pois mytest.R não existe mais. Você já o renomeou.

Este deve ser o único arquivo neste diretório recém-criado. Vamos verificar isso por listando todos os arquivos no diretório atual. list.files () mostra que o diretório contém apenas mytest.R.

Verifique se "mytest.R" existe no diretório de trabalho usando o função file.exists ().

Esses tipos de funções são excessivos para uso interativo. Mas, se você é executando um programa que percorre uma série de arquivos e faz algum processamento em cada um, você vai querer verificar se cada um existe antes de tentar processe.

Acesse informações sobre o arquivo "mytest.R" usando file.info ().

Você pode usar o $ operator --- por exemplo, file.info ("mytest.R") $ mode --- para pegar itens específicos.

Altere o nome do arquivo "mytest.R" para "mytest2.R" usando file.rename ().

Faça uma cópia de "mytest2.R" chamada "mytest3.R" usando file.copy ().

file.copy("mytest2.R", "mytest3.R")

Agora você tem dois arquivos no diretório atual. Isso pode não parecer muito interessante. Mas e se você estivesse trabalhando com dezenas ou milhões de pessoas arquivos? Nesse caso, ser capaz de agir programaticamente em muitos arquivos seria absolutamente necessário. Não se esqueça de que você pode, temporariamente, sair da aula digitando play () e retornar digitando nxt ().

Forneça o caminho relativo para o arquivo "mytest3.R" usando file.path ().

file.path("mytest3.R")

Você pode usar file.path para construir caminhos de arquivo e diretório que sejam independentes do sistema operacional em que seu código R está sendo executado. Passe 'folder1' e 'folder2' *como* argumentos para file.path para criar um nome de caminho independente da plataforma.

file.path("folder1", "folder2")

Dê uma olhada na documentação de dir.create inserindo? Dir.create. Observe o argumento 'recursivo'. Para criar diretórios aninhados, 'recursivo' deve ser definido como TRUE.

Crie um diretório no diretório de trabalho atual chamado "testdir2" e um subdiretório para ele chamado "testdir3", tudo em um comando usando dir.create ()

e file.path ().

dir.create (file.path ('testdir2', 'testdir3'), recursivo = TRUE) fará o

truque. Se você esqueceu o argumento recursivo, o comando pode ter parecido funcionar, mas não criou o diretório aninhado.

Volte para seu diretório de trabalho original usando setwd (). (Lembre-se de que criamos a variável old.dir com o caminho completo para o diretório original de trabalho no início dessas questões.)

Muitas vezes é útil salvar as configurações que você tinha antes de começar um análise e, em seguida, volte a eles no final. Este truque é frequentemente usado dentro funções; você salva, digamos, as configurações de par () com as quais você começou, bagunça um grupo e, em seguida, defina-os de volta para os valores originais no final. Este não é o mesmo como o que fizemos aqui, mas parece semelhante o suficiente para mencionar.

Depois de terminar esta lição, exclua o diretório 'testdir' que você acabou de deixar (e tudo nele)

Não leve nada além de resultados. Não deixe nada além de suposições. Isso soa como 'Take nada além de fotos. Não deixe nada além de pegadas. ' Mas não faz sentido! Certamente nossos leitores podem propor um lema melhor. . .

Nesta lição, você aprendeu como examinar seu espaço de trabalho R e trabalhar com o sistema de arquivos de sua máquina de dentro de R. Obrigado por jogar!

**swirl Lesson 3: Sequences of Numbers**

Nesta lição, você aprenderá a criar sequências de números em R.

A maneira mais simples de criar uma sequência de números em R é usando o método `:`

operador. Digite 1:20 para ver como funciona.

Isso nos deu todos os números inteiros entre (e incluindo) 1 e 20. Também poderíamos usá-lo para criar uma sequência de números reais. Por exemplo, tente pi: 10.

O resultado é um vetor de números reais começando com pi (3.142 ...) e aumentando em incrementos de 1. O limite superior de 10 nunca é atingido, pois o próximo número em nossa sequência seria maior que 10

O que acontece se fizermos 15: 1? Tente descobrir. É contado regressivamente em incrementos de 1! É improvável que queiramos este comportamento, mas mesmo assim é bom saber como isso pode acontecer.

Lembre-se de que, se tiver dúvidas sobre uma função R específica, você pode acessar sua documentação com um ponto de interrogação seguido do nome da função:? Function\_name\_here.

No entanto, no caso de um operador como os dois pontos usados acima, você deve colocar o símbolo entre crases como este:? `:`. (Note o tecla backtick (`) geralmente está localizada no canto superior esquerdo de um teclado, acima a tecla Tab. Se você não tiver uma chave de crase, pode usar aspas normais.)

Frequentemente, desejamos mais controle sobre a sequência que estamos criando do que `:` operador nos dá. A função seq () serve a esse propósito.

O uso mais básico de seq () faz exatamente a mesma coisa que o operador `:`. Tentar seq (1, 20) para ver isso.

Ou talvez não nos importemos com qual é o incremento e queremos apenas uma sequência de 30 números entre 5 e 10. seq (5, 10, comprimento = 30) resolve o problema. Dê uma chance agora e armazene o resultado em uma nova variável chamada my\_seq.

my\_seq <- seq(5,10, length=30) ## Aqui foi determinado o cumprimento de 30 e não a contagem de quantos em quantos

Para confirmar que my\_seq tem comprimento 30, podemos usar a função length ().

Vamos fingir que não sabemos o comprimento de my\_seq, mas queremos gerar um sequência de inteiros de 1 a N, onde N representa o comprimento do vetor my\_seq. Em outras palavras, queremos um novo vetor (1, 2, 3, ...) que seja o mesmo comprimento como my\_seq.

Existem várias maneiras de fazermos isso. Uma possibilidade é combinar o `:`

operador e a função length () como este: 1: length (my\_seq). Experimente.

Outra opção é usar seq (along.with = my\_seq).

No entanto, como é o caso com muitas tarefas comuns, R tem um integrado separado função para este propósito chamada seq\_along (). Digite seq\_along (my\_seq) para vê-lo em açao.

Muitas vezes, existem várias abordagens para resolver o mesmo problema, particularmente em R. Abordagens simples que envolvem menos digitação são geralmente as melhores. É também importante para que seu código seja legível, para que você e outras pessoas possam descobrir o que está acontecendo sem muitos problemas.

Se R tiver uma função incorporada para uma tarefa específica, é provável que essa função seja altamente otimizada para esse propósito e seja sua melhor opção. Conforme você se torna um programador de R mais avançado, você projetará suas próprias funções para realizar tarefas quando não há opções melhores. Exploraremos a escrita de suas próprias funções no futuro lições.

Mais uma função relacionada à criação de sequências de números é rep (), que significa para 'replicar'. Vejamos alguns usos.

Se estivermos interessados ​​em criar um vetor que contenha 40 zeros, podemos usar rep (0, vezes = 40). Experimente.

Se, em vez disso, quisermos que nosso vetor contenha 10 repetições do vetor (0, 1, 2), podemos fazer rep (c (0, 1, 2), vezes = 10). Vá em frente.

Finalmente, digamos que em vez de repetir o vetor (0, 1, 2) indefinidamente, queremos que nosso vetor contenha 10 zeros, depois 10 uns e depois 10 dois. Podemos fazer isso com o argumento `each`. Tente rep (c (0, 1, 2), each = 10).

**swirl Lesson 4: Vectors**

A estrutura de dados mais simples e comum em R é o vetor.

Os vetores vêm em dois sabores diferentes: vetores atômicos e listas. Um vetor atômico contém exatamente um tipo de dados, enquanto uma lista pode conter vários tipos de dados. Exploraremos os vetores atômicos antes de chegar às listas.

Nas lições anteriores, lidamos inteiramente com vetores numéricos, que são um tipo de vetor atômico. Outros tipos de vetores atômicos incluem lógico, caractere, inteiro e complexo. Nesta lição, examinaremos mais de perto os vetores lógicos e de caracteres.

Os vetores lógicos podem conter os valores TRUE, FALSE e NA (para 'não disponível'). Esses valores são gerados como resultado de 'condições' lógicas. Vamos experimentar algumas condições simples.

Primeiro, crie um vetor numérico num\_vect que contém os valores 0,5, 55, -10, e 6

c(0.5, 55, -10, 6)

Lembre-se de que a função c () é usada para criar um vetor. Se você esquecer como usá-lo, use? C para acessar o arquivo de ajuda. Não se esqueça de atribuir o resultado a uma nova variável chamada num\_vect.

num\_vect <- c(0.5, 55, -10, 6)

Agora, crie uma variável chamada tf que obtém o resultado de num\_vect <1, que é lido como 'num\_vect é menor que 1'.

tf <- num\_vect < 1

A declaração num\_vect <1 é uma condição e tf nos diz se cada elemento correspondente do nosso vetor numérico num\_vect satisfaz esta condição.

tf TRUE FALSE TRUE FALSE

O primeiro elemento de num\_vect é 0,5, que é menor que 1 e, portanto, o declaração 0,5 <1 é TRUE. O segundo elemento de num\_vect é 55, que é maior que 1, então a declaração 55 <1 é FALSE. A mesma lógica se aplica ao terceiro e quarto elementos.

Vamos tentar outro. Digite num\_vect> = 6 sem atribuir o resultado a um nova variável.

FALSE TRUE FALSE TRUE

Desta vez, estamos perguntando se cada elemento individual de num\_vect é maior que OR igual a 6. Uma vez que apenas 55 e 6 são maiores ou iguais a 6, o o segundo e o quarto elementos do resultado são VERDADEIROS e o primeiro e o terceiro os elementos são FALSE.

Os símbolos `<` e `> =` nestes exemplos são chamados de 'operadores lógicos'. Outro operadores lógicos incluem `>`, `<=`, `==` para igualdade exata e `! =` para desigualdade.

Se tivermos duas expressões lógicas, A e B, podemos perguntar se pelo menos uma é VERDADEIRA com A | B (lógico 'ou' a.k.a. 'união') ou se ambas são VERDADEIRAS com A & B (lógico 'e' a.k.a. 'intersecção'). Por último,! A é a negação de A e é VERDADEIRO quando A é FALSO e vice-versa.

É uma boa ideia passar algum tempo brincando com várias combinações de esses operadores lógicos até que você se sinta confortável com seu uso. Faremos alguns exemplos aqui para você começar.

Faça o seu melhor para prever o resultado de cada uma das seguintes afirmações. Você pode use lápis e papel para resolvê-los, se for útil. Se você ficar preso, apenas adivinhe e você terá 50% de chance de obter a resposta certa!

(3 > 5) & (4 == 4) Divida este problema em 2 partes. O `&` no meio afirma que AMBAS as peças são verdadeiras. Seu trabalho é descobrir se essa é uma declaração precisa. Nesse caso, toda a condição é TRUE. Se não, é FALSO.

Resposta - FALSE

(TRUE == TRUE) | (TRUE == FALSE) Divida este problema em 2 partes. O `|` no meio afirma que PELO MENOS. UMA das peças é VERDADEIRA. Seu trabalho é descobrir se isso é uma demonstração. Nesse caso, toda a condição é TRUE. Se não, é FALSO.

Resposta - TRUE

((111 >= 111) | !(TRUE)) & ((4 + 1) == 5) Este é complicado. Lembre-se de que o símbolo `!` Nega tudo o que vier depois disso. Também há uma 'ordem de operações' acontecendo aqui. Condições que estão entre parênteses devem ser avaliados primeiro. Então trabalhe seu caminho para fora.

Resposta – TRUE

Não se preocupe se achar que isso é complicado. Eles deveriam ser. Trabalhando com afirmações lógicas em R requer prática, mas seus esforços serão recompensados ​​em lições futuras (por exemplo, subconjuntos e estruturas de controle).

Vetores de caracteres também são muito comuns em R. Aspas duplas são usadas para distinguir objetos de personagem, como no exemplo a seguir.

Crie um vetor de caracteres que contenha as seguintes palavras: "Meu", "nome", "é". Lembre-se de colocar cada palavra em seu próprio conjunto de aspas duplas, para que R saiba que são caracteres cordas. Armazene o vetor em uma variável chamada my\_char.

Digite my\_char <- c ("My", "name", "is") para criar uma nova variável chamada my\_char que contém um vetor de caracteres de comprimento 3. Certifique-se de que as vírgulas que separam as palavras estão FORA das aspas duplas, ou então R pensa que as vírgulas são parte das palavras.

No momento, my\_char é um vetor de caracteres de comprimento 3. Digamos que queremos unir os elementos de my\_char juntos em uma sequência de caracteres contínua (ou seja, um vetor de caráter de comprimento 1). Podemos fazer isso usando a função paste ().

Digite paste (my\_char, collapse = "") agora. Certifique-se de que há um espaço entre o aspas duplas no argumento `recolher`. Você verá por que em um segundo.

O argumento `collapse` para a função paste () diz a R que quando nos juntamos os elementos do vetor de caracteres my\_char, gostaríamos de separá-los com espaços únicos.

Parece que está faltando alguma coisa ... Ah, sim! Seu nome!

Para adicionar (ou 'concatenar') seu nome ao final de my\_char, use a função c ()

assim: c (my\_char, "your\_name\_here"). Coloque seu nome entre aspas onde coloquei "seu\_nome\_aqui". Experimente agora, armazenando o resultado em uma nova variável chamada o meu nome.

Dê uma olhada no conteúdo de my\_name.

Agora, use a função paste () mais uma vez para juntar as palavras em meu\_nome em

uma única sequência de caracteres. Não se esqueça de dizer collapse = ""!

Neste exemplo, usamos a função paste () para recolher os elementos de um vetor de um único caractere. paste () também pode ser usado para juntar os elementos de vários vetores de caracteres.

No caso mais simples, podemos juntar dois vetores de caracteres com comprimento 1 (ou seja, juntar duas palavras). Tente colar ("Hello", "world!", Sep = ""), onde o argumento `sep` diz a R que queremos separar os elementos unidos com um único espaço.

Digite paste ("Hello", "world!", Sep = "") para juntar as duas palavras "Hello" e "mundo", separado por um único espaço. Deve haver um único espaço entre o aspas duplas no argumento `sep` para a função paste ().

Para um exemplo um pouco mais complicado, podemos juntar dois vetores, cada um de comprimento 3. Use paste () para juntar o vetor inteiro 1: 3 com o vetor de caracteres c ("X", "Y", "Z"). Desta vez, use sep = "" para não deixar nenhum espaço entre os elementos unidos.

O que você acha que acontecerá se nossos vetores tiverem comprimentos diferentes? (Dica: falamos sobre isso em uma lição anterior.)

Reciclagem de vetores! Tente colar (LETTERS, 1: 4, sep = "-"), onde LETTERS é um

variável predefinida em R contendo um vetor de caracteres de todas as 26 letras do alfabeto inglês.

Uma vez que o vetor de caracteres LETRAS é mais longo do que o vetor numérico 1: 4, R simplesmente

recicla, ou repete, 1: 4 até que corresponda ao comprimento de LETRAS.

Também vale a pena notar que o vetor numérico 1: 4 é 'coagido' em um vetor de caracteres pela função paste ().

Discutiremos a coerção em outra lição, mas tudo o que realmente significa é que os números 1, 2, 3 e 4 na saída acima não são mais números para R, mas sim

caracteres "1", "2", "3" e "4".

**swirl Lesson 5: Missing Values**

Os valores ausentes desempenham um papel importante nas estatísticas e na análise de dados. Muitas vezes, os valores ausentes não devem ser ignorados, mas sim cuidadosamente estudados para ver se há um padrão ou causa subjacente para a ausência deles.

Qualquer operação envolvendo NA geralmente produz NA como resultado. Para ilustrar, vamos criar um vetor c (44, NA, 5, NA) e atribuí-lo a uma variável x.

Observe que os elementos do vetor resultante que correspondem ao NA os valores em x também são NA.

Para tornar as coisas um pouco mais interessantes, vamos criar um vetor contendo 1000 extrai de uma distribuição normal padrão com y <- rnorm (1000).

Finalmente, vamos selecionar 100 elementos aleatoriamente a partir desses valores de 2.000 (combinando y e z) de modo que não saibamos quantos NAs terminaremos ou quais posições eles ocuparão em nosso vetor final - my\_data <- amostra (c (y, z), 100).

Vamos primeiro fazer a pergunta sobre onde nossos NAs estão localizados em nossos dados. O

A função is.na () nos diz se cada elemento de um vetor é NA. Chame is.na () em my\_data e atribua o resultado a my\_na.

Onde quer que você veja um VERDADEIRO, você sabe que o elemento correspondente de my\_data é NA.

Da mesma forma, em todos os lugares que você vir um FALSO, você conhece o elemento correspondente de

my\_data é um de nossos sorteios aleatórios da distribuição normal padrão.

Em nossa discussão anterior sobre operadores lógicos, introduzimos o operador `==` como um método de teste de igualdade entre dois objetos. Então, você pode pensar que expression my\_data == NA produz os mesmos resultados que is.na (). De uma chance.

A razão pela qual você obteve um vetor de todos os NAs é que NA não é realmente um valor, mas apenas um espaço reservado para uma quantidade que não está disponível. Portanto, o lógico expressão está incompleta e R não tem escolha a não ser retornar um vetor do mesmo length como my\_data que contém todos os NAs.

Não se preocupe se isso for um pouco confuso. A principal lição é ser cauteloso quando usando expressões lógicas a qualquer momento que NAs podem se infiltrar, uma vez que um único valor NA pode descarrilhar a coisa toda.

Vamos tentar aqui. Chame a função sum () em my\_na para contar o total número de TRUEs em my\_na e, portanto, o número total de NAs em my\_data. Não atribua o resultado a uma nova variável.

Muito legal, hein? Finalmente, vamos dar uma olhada nos dados para nos convencer que tudo 'some'. Imprima my\_data no console.

Agora que conhecemos os NAs, vamos examinar um segundo tipo de valor ausente - NaN, que significa 'não um número'. Para gerar NaN, tente dividir (usando um barra) 0 por 0 agora.

Vamos fazer mais um, só por diversão. Em R, Inf significa infinito. O que acontece se você subtrai Inf de Inf?

Irei obter um NaN

**swirl Lesson 6: Subsetting Vectors**

Nesta lição, veremos como extrair elementos de um vetor com base em algumas condições que especificamos.

Por exemplo, podemos estar interessados ​​apenas nos primeiros 20 elementos de um vetor, ou apenas os elementos que não são NA, ou apenas aqueles que são positivos ou correspondem a uma variável específica de interesse. Ao final desta lição, você saberá como lidar com cada um desses cenários.

Eu criei para você um vetor chamado x que contém uma ordem aleatória de 20 números (de uma distribuição normal padrão) e 20 NAs. Digite x agora para ver o que parece.

A maneira como você diz a R que deseja selecionar alguns elementos específicos (ou seja, um 'subconjunto') de um vetor é colocando um 'vetor de índice' entre colchetes imediatamente após o nome do vetor.

Para um exemplo simples, experimente x [1:10] para visualizar os primeiros dez elementos de x.

Os vetores de índice vêm em quatro sabores diferentes - vetores lógicos, vetores de inteiros positivos, vetores de inteiros negativos e vetores de cadeias de caracteres - cada um dos quais abordaremos nesta lição.

Vamos começar indexando com vetores lógicos. Um cenário comum ao trabalhar com dados do mundo real é que queremos extrair todos os elementos de um vetor que são não NA (ou seja, dados ausentes). Lembre-se de que is.na (x) produz um vetor de valores lógicos do mesmo comprimento que x, com TRUEs correspondendo a valores NA em x e FALSEs correspondendo a valores não NA em x.

O que você acha que x [is.na (x)] vai lhe dar? Resposta: Um vetor de todos os NAs

Lembre-se de que is.na (x) nos diz onde os NAs estão em um vetor.

Lembre-se de que `!` Nos dá a negação de uma expressão lógica, então ! is.na (x) pode ser lido como 'não é NA'. Portanto, se quisermos criar um vetor chamado y que contenha todos os valores não NA de x, podemos usar y <- x [! Is.na (x)]. De uma chance.

Agora que isolamos os valores não omissos de x e os colocamos em y, podemos subdividir y como quisermos.

Lembre-se de que a expressão y> 0 nos dará um vetor de valores lógicos do mesmo comprimento que y, com TRUEs correspondendo a valores de y que são maiores que zero e FALSEs correspondendo a valores de y que são menores ou iguais a zero. o que você acha que y [y> 0] vai te dar?

A expressão lógica y> 0 nos dará VERDADEIRO para cada elemento de y que é positivo. Com base nisso, o que você acha que y [y> 0] retornará?

Digite y [y> 0] para ver que obtemos todos os elementos positivos de y, que também são os elementos positivos de nosso vetor original x. Digite y [y> 0] para visualizar apenas os elementos positivos de y.

Você pode se perguntar por que não começamos apenas com x [x> 0] para isolar o positivo elementos de x. Experimente agora para ver por quê.

Uma vez que NA não é um valor, mas sim um espaço reservado para uma quantidade desconhecida, o a expressão NA> 0 avalia como NA. Conseqüentemente, temos um monte de NAs misturados com nossos números positivos quando fazemos isso.

Combinando nosso conhecimento de operadores lógicos com nosso novo conhecimento de subconjunto, poderíamos fazer isso x [! is.na (x) & x> 0]. Experimente.

Nesse caso, solicitamos apenas os valores de x que não estão faltando E são maiores do que zero.

Eu já mostrei como subdividir apenas os primeiros dez valores de x usando x [1:10]. Neste caso, estamos fornecendo um vetor de inteiros positivos dentro de os colchetes, que dizem a R para retornar apenas os elementos de x numerados 1 a 10.

Muitas linguagens de programação usam o que é chamado de 'indexação baseada em zero', o que significa que o primeiro elemento de um vetor é considerado o elemento 0. R usa 'indexação baseada em um', que (você adivinhou!) Significa o primeiro elemento de um vetor é considerado o elemento 1.

Você pode descobrir como dividiríamos o terceiro, o quinto e o sétimo elementos de x? Dica -- Use a função c () para especificar os números dos elementos como um vetor numérico.

x[c(3, 5, 7)]

É importante que, ao usar vetores inteiros para criar um subconjunto de nosso vetor x, mantenhamos com o conjunto de índices {1, 2, ..., 40} uma vez que x tem apenas 40 elementos. O que acontece se pedirmos o elemento zero de x (ou seja, x [0])? De uma chance.

Experimente x [0] para ver o que acontece quando nos referimos ao elemento zero de um vetor, que não é definido em R.

x [0]

Como você pode esperar, não recebemos nada de útil. Infelizmente, R não nos impede de fazer isso. E se pedirmos o 3.000º elemento de x? Experimente.

x [3000]

Novamente, nada de útil, mas R não nos impede de pedir. Este deve ser um conto preventivo. Você deve sempre se certificar de que o que está pedindo é dentro dos limites do vetor com o qual você está trabalhando.

E se estivermos interessados ​​em todos os elementos de x EXCETO o 2 ° e o 10 °? Seria muito tedioso construir um vetor contendo todos os números de 1 a 40, EXCETO 2 e 10.

Felizmente, R aceita índices inteiros negativos. Considerando que x [c (2, 10)] nos dá APENAS o 2º e o 10º elementos de x, x [c (-2, -10)] nos dá todos os elementos de x EXCETO o 2º e os 10 elementos. Experimente x [c (-2, -10)] agora para ver isso

Uma forma abreviada de especificar vários números negativos é colocar o sinal negativo na frente do vetor de números positivos. Digite x [-c (2, 10)] para obter o exatamente o mesmo resultado.

Até agora, cobrimos três tipos de vetores de índice - lógico, inteiro positivo, e inteiro negativo. O único tipo restante exige que apresentemos o conceito de elementos 'nomeados'.

Crie um vetor numérico com três elementos nomeados usando vect <- c (foo = 11, bar = 2, norf = NA).

Quando imprimirmos vect no console, você verá que cada elemento tem um nome. Experimente.

Também podemos obter os nomes de vect passando vect como um argumento para a função names (). Experimente.

names(vect)

Alternativamente, podemos criar um vetor sem nome vect2 com c (11, 2, NA). Faça isso agora.

Então, podemos adicionar o atributo `names` a vect2 após o fato com nomes (vect2) <- c ("foo", "bar", "norf"). Vá em frente.

Agora, vamos verificar se vect e vect2 são iguais, passando-os como argumentos para a função identical ().

identical(vect, vect2)

Na verdade, vect e vect2 são vetores nomeados idênticos.

Agora, de volta à questão de subdividir um vetor por elementos nomeados. Qual dos

seguintes comandos você acha que nos daria o segundo elemento de vect?

Se quisermos o elemento denominado "bar" (ou seja, o segundo elemento de vect), qual comando nos forneceria isso?

Da mesma forma, podemos especificar um vetor de nomes com vect [c ("foo", "bar")]. Tente

Agora você conhece todos os quatro métodos de subconjuntos de dados de vetores. Abordagens diferentes são melhores em cenários diferentes e, em caso de dúvida, experimente!

**swirl Lesson 7: Matrices and Data Frames**

Nesta lição, abordaremos matrizes e data frames (estruturas de dados). Ambos representam tipos de dados 'retangulares', o que significa que são usados ​​para armazenar dados tabulares, com linhas e colunas.

A principal diferença, como você verá, é que as matrizes podem conter apenas uma única classe de dados, enquanto os data frames podem consistir em muitas classes de diferentes dados.

Vamos criar um vetor contendo os números de 1 a 20 usando o operador `:`. Armazene o resultado em uma variável chamada my\_vector.

A função dim () nos diz as 'dimensões' de um objeto. O que acontece se tentarmos - dim(my\_vector)? De uma chance.

Vamos criar um vetor contendo os números de 1 a 20 usando o operador `:`. Armazene o resultado em uma variável chamada my\_vector.

A função dim () nos diz as 'dimensões' de um objeto. O que acontece se escurecermos (my\_vector)? De uma chance.

Claramente, isso não é muito útil! Uma vez que my\_vector é um vetor, ele não tem um Atributo `dim` (então é apenas NULL), mas podemos encontrar seu comprimento usando a função length (). Experimente agora.

Ah! Isso é o que queríamos. Mas, o que acontece se dermos ao my\_vector um `dim` atributo? Vamos tentar. Digite dim (my\_vector) <- c (4, 5).

Tudo bem se esse último comando pareceu um pouco estranho para você. Deveria! O A função dim () permite que você obtenha OU defina o atributo `dim` para um objeto R. Nesse caso, atribuímos o valor c (4, 5) ao atributo `dim` de meu\_vetor.

Use dim (my\_vector) para confirmar que definimos o atributo `dim` corretamente.

Outra maneira de ver isso é chamando a função attribute () em my\_vector. Tente agora.

Assim como na aula de matemática, ao lidar com um objeto bidimensional (pense mesa retangular), o primeiro número é o número de linhas e o segundo é o numero de colunas. Portanto, acabamos de dar ao my\_vector 4 linhas e 5 colunas.

Mas espere! Isso não parece mais um vetor. Bem, não é. Agora é uma matriz. Visualize o conteúdo de my\_vector agora para ver como ele se parece.

Agora, vamos confirmar que é realmente uma matriz usando a função class (). Digite class (my\_vector) para ver o que quero dizer.

Com certeza, meu\_vetor agora é uma matriz. Devemos armazená-lo em uma nova variável que nos ajuda a lembrar o que é. Armazene o valor de my\_vector em uma nova variável chamada my\_matrix.

O exemplo que usamos até agora foi feito para ilustrar o ponto em que uma matriz é simplesmente um vetor atômico com um atributo de dimensão. Um mais direto método de criação da mesma matriz usa a função matrix ().

Abra o arquivo de ajuda para a função matrix () agora usando a função `?`.

?matrix

Agora, olhe a documentação para a função de matriz e veja se você pode descobrir como criar uma matriz contendo os mesmos números (1-20) e dimensões (4 linhas, 5 colunas) chamando a função matrix (). Armazene o resultado em uma variável

chamado my\_matrix2.

my\_matrix2 <- matrix (c (1:20), nrow = 4, ncol = 5)

Finalmente, vamos confirmar se my\_matrix e my\_matrix2 são realmente idênticas. A função idêntica () nos dirá se seus dois primeiros argumentos são iguais. Tente Fora.

Agora, imagine que os números em nossa tabela representam algumas medidas de um experimento clínico, onde cada linha representa um paciente e cada coluna representa uma variável para a qual as medições foram feitas.

Podemos querer rotular as linhas, de modo que saibamos quais números pertencem a cada paciente no experimento. Uma maneira de fazer isso é adicionar uma coluna à matriz, que contém os nomes de todas as quatro pessoas.

Vamos começar criando um vetor de caracteres contendo os nomes de nossos pacientes - Bill, Gina, Kelly e Sean. Lembre-se de que as aspas duplas indicam a R que algo é uma string de caracteres. Armazene o resultado em uma variável chamada pacientes.

pacientes <- c ("Bill", "Gina", "Kelly", "Sean")

Agora usaremos a função cbind () para 'combinar colunas'. Não se preocupe com armazenar o resultado em uma nova variável. Basta chamar cbind () com dois argumentos – o pacientes vetor e my\_matrix.

cbind (patients, my\_matrix)

Há algo suspeito em nosso resultado! Parece que combinar o personagem vetor com nossa matriz de números fez com que tudo fosse colocado em dobro citações. Isso significa que ficamos com uma matriz de cadeias de caracteres, o que não é bom.

Se você se lembra do início desta lição, eu disse que as matrizessó pode conter UMA classe de dados. Portanto, quando tentamos combinar um vetor de caracteres com uma matriz numérica, R foi forçado a 'coagir' os números a caracteres, daí as aspas duplas.

Então, ainda ficamos com a questão de como incluir os nomes de nossos pacientes na tabela sem destruir a integridade de nossos dados numéricos. Tentar o seguinte - my\_data <- data.frame (patients, my\_matrix)

Parece que a função data.frame () nos permitiu armazenar nosso vetor de caracteres de nomes ao lado de nossa matriz de números. Isso é exatamente o que esperávamos!

Nos bastidores, a função data.frame () recebe qualquer número de argumentos e retorna um único objeto da classe `data.frame` que é composto do original objetos.

Vamos confirmar isso chamando a função class () em nossos dados recém-criados.

classe (my\_data)

Também é possível atribuir nomes às linhas e colunas individuais de um quadro de dados, o que apresenta outra maneira possível de determinar em qual linha de valores nossa mesa pertence a cada paciente.

No entanto, como já resolvemos esse problema, vamos resolver um problema diferente atribuindo nomes às colunas de nosso quadro de dados para que saibamos que tipo de medição cada coluna representa.

Como temos seis colunas (incluindo nomes de pacientes), primeiro precisamos criar um vetor contendo um elemento para cada coluna. Crie um vetor de caracteres chamado cnames que contém os seguintes valores (em ordem) - ("patient", "age", "weight", "bp", "rating", "test")

cnames <- c("patient", "age", "weight", "bp", "rating", "test")

Agora, use a função colnames () para definir o atributo `colnames` para nosso quadro de dados. Isso é semelhante à maneira como usamos a função dim () anteriormente nesta lição.

colnames (my\_data) <- cnames

Nesta lição, você aprendeu o básico para trabalhar com duas estruturas de dados comuns e muito importantes - matrizes e quadros de dados. Há muito mais para aprender e abordaremos tópicos mais avançados, principalmente em relação aos dados, em aulas futuras.

**Semana 2**

**swirl Lesson 1: Logic**

A criação de expressões lógicas requer operadores lógicos. Você provavelmente está familiarizado com operadores aritméticos como `+`, `-`,` \* `e` / `. O primeiro operador lógico que vamos discutir é o operador de igualdade, representado por dois sinais de igual `==`. Use o operador de igualdade abaixo para descobrir se TRUE é igual a TRUE.

Assim como a aritmética, as expressões lógicas podem ser agrupadas por parênteses para que a expressão inteira (VERDADEIRO == VERDADEIRO) == VERDADEIRO seja avaliada como VERDADEIRA.

Para testar esta propriedade, tente avaliar (FALSE == TRUE) == FALSE.

O operador de igualdade também pode ser usado para comparar números. Use `==` para ver se 6 é igual a 7.

A expressão anterior é avaliada como FALSE porque 6 é menor que 7. Felizmente, existem operadores de desigualdade que nos permitem testar se um valor é menor que ou maior do que outro valor.

O operador menor que `<` testa se o número no lado esquerdo do operador (chamado de operando esquerdo) é menor que o número do lado direito de o operador (chamado de operando certo). Escreva uma expressão para testar se 6 é menos de 7.

Há também um operador menor ou igual a `<=` que testa se o operando esquerdo é menor ou igual ao operando direito. Escreva uma expressão para testar se 10 é menor ou igual a 10.

Lembre-se de que existem os correspondentes maiores que `>` e maior que ou igual a `> =` operadores.

O próximo operador que discutiremos é o operador 'diferente de' representado por `! =`. Diferente testa se dois valores são desiguais, portanto TRUE! = FALSE avalia para TRUE. Como o operador de igualdade, `! =` Também pode ser usado com números. Tentar escrever uma expressão para ver se 5 não é igual a 7.

Para negar expressões booleanas, você pode usar o operador NOT. A ponto de exclamação `!` fará com que! TRUE (diga: não verdadeiro) seja avaliado como FALSE e ! FALSE (diga: não falso) para avaliar como TRUE. Tente usar o operador NOT e o operador igual a para encontrar o oposto de 5 é igual a 7.

Esta expressão pode ser um pouco complicada, então pense em negar a expressão 5 == 7 (tudo que você precisa é um ponto de exclamação na frente).

Vamos fazer uma revisão. O operador igual `==` testa se dois booleanos valores ou números são iguais, o operador diferente de `! =` testa se dois valores booleanos ou números são desiguais, e o operador NOT `!` nega a lógica expressões para que as expressões TRUE se tornem FALSE e as expressões FALSE se tornem VERDADEIRO.

Em algum ponto, você pode precisar examinar as relações entre vários expressões. É aqui que o operador AND e o operador OR entram.

Vejamos como o operador AND funciona. Existem dois operadores AND em R, `&` e `&&`. Ambos os operadores funcionam de forma semelhante, se os operandos direito e esquerdo de AND são ambos TRUE, a expressão inteira é TRUE, caso contrário, é FALSE. Por exemplo, TRUE & TRUE é avaliado como TRUE. Tente digitar FALSE & FALSE para como ele é avaliado.

Você pode usar o operador `&` para avaliar o AND em um vetor. A versão `&&` de AND avalia apenas o primeiro membro de um vetor. Vamos testar os dois para praticar.

Digite a expressão TRUE & c (TRUE, FALSE, FALSE).

O que acontece neste caso é que o operando esquerdo `TRUE` é reciclado em cada elemento no vetor do operando certo. Este é o equivalente declaração como c (VERDADEIRO, VERDADEIRO, VERDADEIRO) & c (VERDADEIRO, FALSO, FALSO).

Agora, digitaremos a mesma expressão, mas usaremos o operador `&&`. Digite a expressão TRUE && c (TRUE, FALSE, FALSE).

Nesse caso, o operando esquerdo só é avaliado com o primeiro membro do operando à direita (o vetor). O resto dos elementos do vetor não são avaliado em tudo nesta expressão.

O operador OR segue um conjunto semelhante de regras. A versão `|` de OR avaliaOR em todo um vetor, enquanto a versão `||` de OR avalia apenas o primeiro membro de um vetor.

Uma expressão usando o operador OR será avaliada como TRUE se o operando esquerdo ou o operando certo é TRUE. Se ambos forem TRUE, a expressão será avaliada como TRUE, no entanto, se nenhum for TRUE, a expressão será FALSE.

Vamos testar a versão vetorizada do operador OR. Digite a expressão | VERDADEIRO | c (VERDADEIRO, FALSO, FALSO).

Agora vamos experimentar a versão não vetorizada do operador OR. Digite a expressão TRUE || c (VERDADEIRO, FALSO, FALSO).

Os operadores lógicos podem ser encadeados exatamente como os operadores aritméticos. A expressões: `6! = 10 && FALSE && 1> = 2` ou` TRUE || 5 <9,3 || FALSE` são perfeitamente normal de ver.

Como você deve se lembrar, a aritmética tem uma ordem de operações, assim como as expressões lógicas. Todos os operadores AND são avaliados antes dos operadores OR. Vamos olhar para um exemplo de caso ambíguo. Digite: 5> 8 || 6! = 8 && 4> 3,9

Vamos examinar a ordem das operações no caso acima. Primeiro, os operandos esquerdo e direito do operador AND são avaliados. 6 não é igual a 8, 4 é maior que 3,9, portanto, ambos os operandos são TRUE, então a expressão resultante `TRUE && TRUE` é avaliada como TRUE. Em seguida, o operando esquerdo do operador OR é avaliado: 5 não é maior que 8, portanto, toda a expressão é reduzida a FALSE || VERDADEIRO. Uma vez que o operando direito desta expressão é TRUE, a expressão inteira é avaliada como VERDADEIRO.

A função isTRUE () leva um argumento. Se esse argumento for avaliado como TRUE, o a função retornará TRUE. Caso contrário, a função retornará FALSE. Tente usar esta função digitando: isTRUE (6> 4)

A função identical () retornará TRUE se os dois objetos R passados ​​para ela como os argumentos são idênticos. Experimente a função idêntica () digitando: identical ('gêmeos', 'gêmeos')

Você também deve estar ciente da função xor (), que recebe dois argumentos. A função xor () significa OR exclusivo. Se um argumento for avaliado como TRUE e um argumento for avaliado como FALSO, então esta função retornará VERDADEIRO, caso contrário, retornará FALSO. Experimente a função xor () digitando: xor (5 == 6,! FALSE)

5 == 6 avalia como FALSE,! FALSE avalia como TRUE, então xor (FALSE, TRUE) avalia como TRUE. Por outro lado, se o primeiro argumento foi alterado para 5 == 5 e o segundo argumento não foi alterado, então ambos os argumentos seriam TRUE, então xor (TRUE, TRUE) seria avaliado como FALSE.

Para as próximas perguntas, vamos precisar criar um vetor de inteiros chamado ints. Crie este vetor digitando: ints <- sample (10)

Agora, basta exibir o conteúdo de ints.

O vetor `ints` é uma amostra aleatória de inteiros de 1 a 10 sem substituição. Digamos que queremos fazer algumas perguntas lógicas sobre o conteúdo de ints. Se digitarmos ints> 5, obteremos um vetor lógico correspondente a se cada elemento de ints é maior que 5. Tente digitar: ints > 5

Podemos usar o vetor lógico resultante para fazer outras perguntas sobre ints. O qual função () pega um vetor lógico como argumento e retorna os índices de o vetor que é VERDADEIRO. Por exemplo qual (c (VERDADEIRO, FALSO, VERDADEIRO)) retornaria o vetor c (1, 3).

Use a função which () para encontrar os índices de ints maiores que 7.

Qual dos seguintes comandos produziria os índices dos elementos em ints que são menores ou iguais a 2?

1: qual (ints <= 2)

2: ints <= 2

3: qual (ints <2)

4: ints <2

Como a função which (), as funções any () e all () tomam vetores lógicos como seu argumento. A função any () retornará TRUE se um ou mais dos elementos no vetor lógico for TRUE. A função all () retornará TRUE se cada elemento no vetor lógico é TRUE.

Use a função any () para ver se algum dos elementos de ints é menor que zero.

any(ints < 0)

Use a função all () para ver se todos os elementos de ints são maiores que zero.

all(ints > 0)

**swirl Lesson 2: Functions**

As funções são um dos blocos de construção fundamentais da linguagem R. Eles são pequenos pedaços de código reutilizável que podem ser tratados como qualquer outro objeto R.

Se você trabalhou em qualquer outra parte deste curso, provavelmente já usou algumas funções. As funções geralmente são caracterizadas pelo nome da função seguido por parênteses.

Vamos tentar usar algumas funções básicas apenas por diversão. A função Sys.Date () retorna uma string que representa a data de hoje. Digite Sys.Date () abaixo e veja o que acontece.

A maioria das funções em R retorna um valor. Funções como Sys.Date () retornam um valor com base no ambiente do seu computador, enquanto outras funções manipulam os dados de entrada para calcular um valor de retorno.

A função mean () pega um vetor de números como entrada e retorna a média de todos os números no vetor de entrada. As entradas para funções são freqüentemente chamadas de argumentos. Fornecer argumentos para uma função também é às vezes chamado de passar argumentos para essa função. Os argumentos que você deseja passar para uma função vão para dentro os parênteses da função. Tente passar o argumento c (2, 4, 5) para a média () função.

As funções geralmente aceitam argumentos que são variáveis ​​nas quais a função opera. Por exemplo, a função mean () toma um vetor como argumento, como no caso da média (c (2,6,8)). A função mean () então soma todos os números no vetor e divide essa soma pelo comprimento do vetor.

Na pergunta a seguir, você será solicitado a modificar um script que aparecerá assim que você passar desta questão. Quando terminar de modificar o script, salve suas alterações no script e digite submit () e o script será avaliado. Haverá alguns comentários no script que será aberto, portanto, certifique-se de lê-los!

Agora que você criou sua primeira função, vamos testá-la! Modelo: boring\_function ('Minha primeira função!'). Se sua função funcionar, ela deve apenas retornar a string: 'Minha primeira função!'

Parabéns por escrever sua primeira função. Ao escrever funções, você pode obter uma visão séria de como o R funciona. Como John Chambers, o criador do R disse uma vez:

Para entender os cálculos em R, dois slogans são úteis: 1. Tudo o que existe é um objeto. 2. Tudo o que acontece é uma chamada de função.

Se você quiser ver o código-fonte de qualquer função, basta digitar o nome da função sem argumentos ou parênteses. Vamos tentar fazer isso com a função que você acabou de criar. Digite: boring\_function para visualizar seu código-fonte.

É hora de fazer uma função mais útil! Vamos replicar a funcionalidade da função mean () criando uma função chamada: my\_mean (). Lembre-se de que, para calcular a média de todos os números de um vetor, você encontra a soma de todos os números do vetor e, a seguir, divide essa soma pelo número de números do vetor.

**swirl Lesson 3: Dates and Times**

R tem uma maneira especial de representar datas e horas, o que pode ser útil se você estiver trabalhando com dados que mostram como algo muda ao longo do tempo (ou seja, dados de série temporal) ou se seus dados contêm alguma outra informação temporal, como datas de nascimento.

As datas são representadas pela classe 'Date' e os horários pelas classes 'POSIXct' e 'POSIXlt'. Internamente, as datas são armazenadas como o número de dias desde 01/01/1970 e as horas são armazenadas como o número de segundos desde 1970-01-01 (para 'POSIXct') ou uma lista de segundos, minutos, horas, etc. (para 'POSIXlt').

Vamos começar usando d1 <- Sys.Date () para obter a data atual e armazená-la no variável d1. (Essa é a letra 'd' e o número 1.)

Use a função class () para confirmar que d1 é um objeto Date.

Podemos usar a função unclass () para ver como d1 se parece internamente. Experimente.

Este é o número exato de dias desde 01/01/1970!

No entanto, se você imprimir d1 no console, obterá a data de hoje - ANO MÊS DIA. De uma chance.

E se precisarmos fazer referência a uma data anterior a 01/01/1970? Crie uma variável d2 contendo as.Date ("1969-01-01").

Agora use unclass () novamente para ver como d2 se parece internamente.

Como você deve ter previsto, você obtém um número negativo. Neste caso, é -365, desde 1969-01-01 é exatamente um ano civil (ou seja, 365 dias) ANTES de 1970-01-01.

Agora, vamos dar uma olhada em como R armazena os tempos. Você pode acessar a data e hora atuais usando a função Sys.time () sem argumentos. Faça isso e armazene o resultar em uma variável chamada t1.

Conforme mencionado anteriormente, POSIXct é apenas uma das duas maneiras em que R representa as informações de tempo. (Você pode ignorar o segundo valor acima, POSIXt, que funciona apenas como uma linguagem comum entre POSIXct e POSIXlt.) Use unclass () para ver como t1 se parece internamente - o (grande) número de segundos desde o início de 1970

Por padrão, Sys.time () retorna um objeto da classe POSIXct, mas podemos forçar o resultado para POSIXlt com as.POSIXlt (Sys.time ()). Experimente e armazene o resultado em t2.

Verifique a classe de t2

Agora veja seu conteúdo.

O formato impresso de t2 é idêntico ao de t1. Agora desclassifique () t2 para ver como é diferente internamente.

t2, como todos os objetos POSIXlt, é apenas uma lista de valores que constituem a data e a hora. Use str (unclass (t2)) para ter uma visão mais compacta.

Se, por exemplo, quisermos apenas os minutos do tempo armazenado em t2, podemos acessá-los com t2 $ min. De uma chance.

Agora que exploramos todos os três tipos de objetos de data e hora, vamos dar uma olhada em algumas funções que extraem informações úteis de qualquer um desses objetos - dias da semana (), meses () e trimestres ().

A função weekdays () retornará o dia da semana de qualquer objeto de data ou hora.

Experimente em d1, que é o objeto Date que contém a data de hoje.

A função months () também funciona em qualquer objeto de data ou hora. Experimente em t1, que é o objeto POSIXct que contém a hora atual (bem, era a hora atual quando você o criou).

A função quarters () retorna o trimestre do ano (Q1-Q4) a partir de qualquer data ou objeto de tempo. Tente em t2, que é o objeto POSIXlt que contém a hora em que você o criou.

Freqüentemente, as datas e horas em um conjunto de dados estarão em um formato que R não reconhece. A função strptime () pode ser útil nessa situação.

strptime () converte vetores de caracteres em POSIXlt. Nesse sentido, é semelhante a as.POSIXlt (), exceto que a entrada não precisa estar em um formato específico (AAAA-MM-DD).

Para ver como funciona, armazene a seguinte sequência de caracteres em uma variável chamada t3: "17 de outubro de 1986 08:24" (com as aspas).

A função weekdays () retornará o dia da semana de qualquer objeto de data ou hora.

Experimente em d1, que é o objeto Date que contém a data de hoje.

A função months () também funciona em qualquer objeto de data ou hora. Experimente em t1, que é o objeto POSIXct que contém a hora atual (bem, era a hora atual quando você o criou).

A função quarters () retorna o trimestre do ano (Q1-Q4) a partir de qualquer data ou objeto de tempo. Tente em t2, que é o objeto POSIXlt que contém a hora em que você o criou.

Frequentemente, as datas e horas em um conjunto de dados estarão em um formato que R não reconhece. A função strptime () pode ser útil nessa situação.

strptime () converte vetores de caracteres em POSIXlt. Nesse sentido, é semelhante a as.POSIXlt (), exceto que a entrada não precisa estar em um formato específico (AAAA-MM-DD).

Para ver como funciona, armazene a seguinte sequência de caracteres em uma variável chamada t3: "17 de outubro de 1986 08:24" (com as aspas).

Agora, use strptime (t3, "% B% d,% Y% H:% M") para ajudar R a converter nosso objeto de data / hora em um formato que ele entenda. Atribua o resultado a uma nova variável chamada t4. (Você deve consultar a documentação de strptime () se quiser saber mais sobre como funciona.)

Imprima o conteúdo de t4

Esse é o formato que esperamos. Agora, vamos verificar sua classe ()

Finalmente, há uma série de operações que você pode realizar em datas e horas, incluindo operações aritméticas (+ e -) e comparações (<, ==, etc.)

A variável t1 contém a hora em que você a criou (lembre-se de que você usou Sys.time ()). Confirme se já passou algum tempo desde que você criou t1 usando o operador 'maior que' para compará-lo com a hora atual: Sys.time ()> t1

Então sabemos que já passou algum tempo, mas quanto? Tente subtrair t1 da hora atual usando Sys.time () - t1. Não se esqueça dos parênteses no final de Sys.time (), pois é uma função.

A função weekdays () retornará o dia da semana de qualquer objeto de data ou hora.

Experimente em d1, que é o objeto Date que contém a data de hoje.

A função months () também funciona em qualquer objeto de data ou hora. Experimente em t1, que é o objeto POSIXct que contém a hora atual (bem, era a hora atual quando você o criou).

A função quarters () retorna o trimestre do ano (Q1-Q4) a partir de qualquer data ou objeto de tempo. Tente em t2, que é o objeto POSIXlt que contém a hora em que você o criou.

Frequentemente, as datas e horas em um conjunto de dados estarão em um formato que R não reconhece. A função strptime () pode ser útil nessa situação.

strptime () converte vetores de caracteres em POSIXlt. Nesse sentido, é semelhante a as.POSIXlt (), exceto que a entrada não precisa estar em um formato específico (AAAA-MM-DD).

Para ver como funciona, armazene a seguinte sequência de caracteres em uma variável chamada t3: "17 de outubro de 1986 08:24" (com as aspas).

Agora, use strptime (t3, "% B% d,% Y% H:% M") para ajudar R a converter nosso objeto de data / hora em um formato que ele entenda. Atribua o resultado a uma nova variável chamada t4.

(Você deve consultar a documentação de strptime () se quiser saber mais sobre como funciona.)

Imprima o conteúdo de t4

Esse é o formato que esperamos. Agora, vamos verificar sua classe ()

Finalmente, há uma série de operações que você pode realizar em datas e horas, incluindo operações aritméticas (+ e -) e comparações (<, ==, etc.)

A variável t1 contém a hora em que você a criou (lembre-se de que você usou

Sys.time ()). Confirme se já passou algum tempo desde que você criou t1 usando o operador 'maior que' para compará-lo com a hora atual: Sys.time ()> t1

Então sabemos que já passou algum tempo, mas quanto? Tente subtrair t1 da hora atual usando Sys.time () - t1. Não se esqueça dos parênteses no final de Sys.time (), pois é uma função

A mesma linha de pensamento se aplica à adição e aos outros operadores de comparação.

Se você quiser mais controle sobre as unidades ao encontrar a diferença de tempo acima, você pode usar difftime (), que permite que você especifique um parâmetro de 'unidades'

Use difftime (Sys.time (), t1, unidades = 'dias') para encontrar a quantidade de tempo em DAYS que passou desde que você criou t1

Nesta lição, você aprendeu como trabalhar com datas e horas em R. Enquanto importante entender o básico, se você estiver trabalhando com datas e horas, você pode querer verificar o pacote de lubrificantes de Hadley Wickham.

**swirl Lesson 1: lapply and sapply**

Essas funções poderosas, junto com seus parentes próximos (vapply () e tapply (), entre outros) oferece um meio conciso e conveniente de implementar a estratégia Split-Apply-Combine para análise de dados.

Cada uma das \* funções de aplicação Dividirá alguns dados em partes menores, APLICARÁ uma função a cada parte e, em seguida, COMBINARÁ os resultados. Uma discussão mais detalhada dessa estratégia pode ser encontrada no artigo de Hadley Wickham's Journal of Statistical Software intitulado 'A estratégia dividir-aplicar-combinar para análise de dados'.

Ao longo desta lição, usaremos o conjunto de dados Flags do UCI Machine Learning Repository. Este conjunto de dados contém detalhes de várias nações e suas bandeiras.

Mais informações podem ser encontradas aqui: http://archive.ics.uci.edu/ml/datasets/Flags

Vamos começar para que você tenha uma ideia de como essas funções especiais funcionam!

Eu armazenei o conjunto de dados em uma variável chamada sinalizadores. Digite head (flags) para visualizar as primeiras seis linhas (ou seja, o 'head') do conjunto de dados.

Pode ser necessário rolar para cima para ver toda a saída. Agora, vamos verificar as dimensões do conjunto de dados usando dim (sinalizadores).

Isso nos diz que existem 194 linhas ou observações e 30 colunas ou variáveis. Cada observação é um país e cada variável descreve alguns

característica desse país ou de sua bandeira. Para abrir uma descrição mais completa do conjunto de dados em um arquivo de texto separado, digite viewinfo () quando voltar ao prompt (>).

Como acontece com qualquer conjunto de dados, gostaríamos de saber em que formato as variáveis ​​foram armazenadas. Em outras palavras, qual é a 'classe' de cada variável? O que acontece se fizermos aula (sinalizadores)? Experimente.

Isso apenas nos diz que todo o conjunto de dados é armazenado como um 'data.frame', o que não responde à nossa pergunta. O que realmente precisamos é chamar a função class () em cada coluna individual. Embora possamos fazer isso manualmente (ou seja, uma coluna por vez), é muito mais rápido se pudermos automatizar o processo. Parece um loop!

A função lapply () recebe uma lista como entrada, aplica uma função a cada elemento da lista e retorna uma lista do mesmo comprimento da original. Como um quadro de dados é na verdade apenas uma lista de vetores (você pode ver isso com as.list (sinalizadores)), podemos usar lapply () para aplicar a função class () a cada coluna do conjunto de dados sinalizadores. Vamos ver em ação!

Digite cls\_list <- lapply (sinalizadores, classe) para aplicar a função class () a cada coluna do conjunto de dados sinalizadores e armazena o resultado em uma variável chamada cls\_list. Observe que você apenas fornece o nome da função que deseja aplicar (ou seja, classe), sem os parênteses usuais após ela.

Digite cls\_list para ver o resultado.

O 'l' em 'lapply' significa 'lista'. Digite class (cls\_list) para confirmar que lapply () retornou uma lista.



Como esperado, obtivemos uma lista de comprimento 30 - um elemento para cada variável / coluna.

A saída seria consideravelmente mais compacta se pudéssemos representá-la como um vetor em vez de uma lista.

Você deve se lembrar de uma lição anterior que as listas são mais úteis para armazenar várias classes de dados. Neste caso, uma vez que cada elemento da lista retornado por lapply () é um vetor de caracteres de comprimento um (ou seja, "inteiro" e "vetor"), cls\_list pode ser simplificado para um vetor de caracteres. Para fazer isso manualmente, digite

as.character (cls\_list).

sapply () permite automatizar esse processo chamando lapply () nos bastidores, mas tentando simplificar (daí o 's' em 'sapply') o resultado para você. Use sapply () da mesma forma que você usou lapply () para obter a classe de cada coluna do conjunto de dados de sinalizadores e armazenar o resultado em cls\_vect. Se precisar de ajuda, digite?

Sapply para exibir a documentação.

Use class (cls\_vect) para confirmar que sapply () simplificou o resultado para um vetor de caracteres.

Em geral, se o resultado for uma lista em que cada elemento tem o comprimento um, sapply () retorna um vetor. Se o resultado for uma lista onde cada elemento é um vetor do mesmo comprimento (> 1), sapply () retorna uma matriz. Se sapply () não consegue descobrir as coisas, então ele apenas retorna uma lista, não diferente do que lapply () lhe daria.

Vamos praticar usando lapply () e sapply () um pouco mais!

As colunas 11 a 17 do nosso conjunto de dados são variáveis ​​indicadoras, cada uma representando um cor diferente. O valor da variável indicadora é 1 se a cor está presente na bandeira de um país e 0 caso contrário.

Portanto, se quisermos saber o número total de países (em nosso conjunto de dados) com, por exemplo, a cor laranja em sua bandeira, podemos simplesmente adicionar todos os 1s e 0s na coluna 'laranja'. Experimente sum (sinalizadores $ laranja) para ver isso.

Agora queremos repetir esta operação para cada uma das cores registradas no conjunto de dados.

Primeiro, use flag\_colors <- flags [, 11:17] para extrair as colunas contendo os dados de cores e armazene-as em um novo quadro de dados chamado flag\_colors. (Observe a vírgula antes de 11:17. Este comando de subconjunto diz a R que queremos todas as linhas, mas apenas colunas 11 a 17.)

Use a função head () para ver as primeiras 6 linhas de flag\_colors.

Para obter uma lista contendo a soma de cada coluna de flag\_colors, chame lapply () função com dois argumentos. O primeiro argumento é o objeto sobre o qual estamos fazendo um loop (ou seja, flag\_colors) e o segundo argumento é o nome da função que desejamos aplicar a cada coluna (ou seja, soma). Lembre-se de que o segundo argumento é apenas o nome da função sem parênteses, etc.

Isso nos diz que das 194 sinalizações em nosso conjunto de dados, 153 contêm a cor vermelha, 91 contém verde, 99 contém azul e assim por diante.

O resultado é uma lista, já que lapply () sempre retorna uma lista. Cada elemento desta lista tem comprimento um, então o resultado pode ser simplificado para um vetor chamando sapply () em vez de lapply (). Tente agora.

Use sapply (flag\_colors, sum) para retornar um vetor de somas de coluna.

Talvez seja mais informativo encontrar a proporção de bandeiras (de 194)

contendo cada cor. Como cada coluna é apenas um grupo de 1s e 0s, a média aritmética de cada coluna nos dará a proporção de 1s. (Se não estiver claro por que, pense em uma situação mais simples onde você tem três 1s e dois 0s - (1 + 1 + 1 + 0 + 0) / 5 = 3/5 = 0,6).

Use sapply () para aplicar a função mean () a cada coluna de flag\_colors. Lembre-se de que o segundo argumento para sapply () deve apenas especificar o nome da função (ou seja, média) que você deseja aplicar.

Nos exemplos que vimos até agora, sapply () foi capaz de simplificar o resultado para vetor. Isso ocorre porque cada elemento da lista retornado por lapply () era um vetor de comprimento um. Lembre-se de que sapply (), em vez disso, retorna uma matriz quando cada elemento da lista retornada por lapply () é um vetor do mesmo comprimento (> 1).

Para ilustrar isso, vamos extrair as colunas 19 a 23 do conjunto de dados de sinalizadores e armazene o resultado em um novo quadro de dados chamado flag\_shapes. flag\_shapes <-flags [, 19:23] fará isso.

Cada uma dessas colunas (ou seja, variáveis) representa o número de vezes que uma forma ou desenho específico aparece na bandeira de um país. Estamos interessados ​​no número mínimo e máximo de vezes que cada forma ou desenho aparece.

A função range () retorna o mínimo e o máximo de seu primeiro argumento, que deve ser um vetor numérico. Use lapply () para aplicar a função de intervalo a cada coluna de flag\_shapes. Não se preocupe em armazenar o resultado em uma nova variável. Agora, sabemos que lapply () sempre retorna uma lista.

Faça a mesma operação, mas usando sapply () e armazene o resultado em uma variável chamada shape\_mat.

shape\_mat <- sapply (flag\_shapes, range) irá aplicar a função range () a cada coluna de flag\_shapes e armazenar o resultado em shape\_mat.

Visualize o conteúdo de shape\_mat.

Cada coluna de shape\_mat fornece o número mínimo (linha 1) e máximo (linha 2) de vezes que sua forma respectiva aparece em sinalizadores diferentes.

class (shape\_mat) retorna a classe de shape\_mat.

Como vimos, sapply () sempre tenta simplificar o resultado dado por lapply ().

Isso foi bem-sucedido em cada um dos exemplos que examinamos até agora. Vejamos um exemplo em que sapply () não consegue descobrir como simplificar o resultado e, portanto, retorna uma lista, não diferente de lapply ().

Quando dado um vetor, a função unique () retorna um vetor com todos os elementos duplicados removidos. Em outras palavras, unique () retorna um vetor de apenas os elementos 'únicos'. Para ver como funciona, experimente exclusivo (c (3, 4, 5, 5, 5, 6, 6)).

Queremos saber os valores únicos para cada variável no conjunto de dados de sinalizadores.

Para fazer isso, use lapply () para aplicar a função unique () a cada coluna no conjunto de dados de sinalizadores, armazenando o resultado em uma variável chamada unique\_vals.

Visto que unique\_vals é uma lista, você pode usar o que aprendeu para determinar o comprimento de cada elemento de unique\_vals (ou seja, o número de valores únicos para cada variável). Simplifique o resultado, se possível. Dica: aplique a função length () a cada elemento de unique\_vals.

Aplique a função length () a cada elemento da lista unique\_vals usando sapply (). Lembre-se de que não há parênteses após o nome da função que você está aplicando (ou seja, comprimento).

O fato de que os elementos da lista unique\_vals são todos vetores de \* diferentes \* comprimento representa um problema para sapply (), uma vez que não há uma maneira óbvia de simplificar o resultado.

Use sapply () para aplicar a função unique () a cada coluna do conjunto de dados sinalizadores para ver se você obtém a mesma lista não simplificada que obteve em lapply ().

Ocasionalmente, você pode precisar aplicar uma função que ainda não está definida, portanto exigindo que você escreva o seu próprio. Escrever funções em R está além do escopo desta lição, mas vamos ver um exemplo rápido de como você pode fazer isso no contexto de funções de loop.

Finja que você está interessado apenas no segundo item de cada elemento da lista unique\_vals que você acabou de criar. Uma vez que cada elemento do unique\_vals lista é um vetor e não temos conhecimento de nenhuma função incorporada em R que retorna o segundo elemento de um vetor, construiremos nossa própria função.

lapply (unique\_vals, function (elem) elem [2]) retornará uma lista contendo o segundo item de cada elemento da lista unique\_vals. Observe que nossa função recebe um argumento, elem, que é apenas uma 'variável fictícia' que assume o valor de cada elemento de unique\_vals, por sua vez.

A única diferença entre os exemplos anteriores e este é que estamos definindo e usando nossa própria função na chamada de lapply (). Nossa função não tem nome e desaparece assim que lapply () termina de usá-la. As chamadas 'funções anônimas' podem ser muito úteis quando uma das funções integradas de R não é uma opção.

**swirl Lesson 2: vapply and tapply**

Na última lição, você aprendeu sobre os dois membros mais fundamentais da família de funções de R's \* apply: lapply () e sapply (). Ambos recebem uma lista como entrada, aplicam uma função a cada elemento da lista e, a seguir, combinam e retornam o resultado. lapply () sempre retorna uma lista, enquanto sapply () tenta simplificar o resultado.

Nesta lição, você aprenderá como usar vapply () e tapply (), cada um dos quais serve a um propósito muito específico dentro da metodologia Dividir-Aplicar-Combinar. Para consistência, usaremos o mesmo conjunto de dados que usamos na lição 'lapply and sapply'.

O conjunto de dados Flags do UCI Machine Learning Repository contém detalhes de várias nações e suas bandeiras. Mais informações podem ser encontradas aqui: http://archive.ics.uci.edu/ml/datasets/Flags

Eu armazenei os dados em uma variável chamada flags. Se já faz um tempo que você concluiu a lição 'lapply and sapply', você pode querer se familiarizar novamente com os dados usando funções como dim (), head (), str () e summary () quando você retorna ao prompt (>). Você também pode digitar viewinfo () no prompt para trazer alguma documentação para o conjunto de dados. Vamos começar!

Como você viu na última lição, a função unique () retorna um vetor dos valores únicos contidos no objeto passado para ela. Portanto, sapply (sinalizadores, exclusivo) retorna uma lista contendo um vetor de valores exclusivos para cada coluna do conjunto de dados de sinalizadores. Tente novamente agora.

E se você tivesse esquecido como o unique () funciona e erroneamente pensasse que ele retorna o \* número \* de valores únicos contidos no objeto passado para ele? Então você pode ter esperado incorretamente (sinalizadores, únicos) para retornar um vetor numérico, já que cada elemento da lista retornada conteria um único número e sapply () poderia então simplificar o resultado para um vetor.

Ao trabalhar de forma interativa (no prompt), isso não é um grande problema, uma vez que

você verá o resultado imediatamente e reconhecerá rapidamente seu erro. No entanto, ao trabalhar de forma não interativa (por exemplo, escrever suas próprias funções), um mal-entendido podem passar despercebido e causar resultados incorretos posteriormente.

Portanto, você pode querer ser mais cuidadoso e é aí que vapply () é útil.

Enquanto sapply () tenta 'adivinhar' o formato correto do resultado, vapply () permite que você especifique explicitamente. Se o resultado não corresponder ao formato que você especificar, vapply () lançará um erro, fazendo com que a operação pare. Isso pode evitar problemas significativos em seu código que podem ser causados ​​pela obtenção de valores de retorno inesperados de sapply ().

Tente vapply (flags, unique, numeric (1)), que diz que você espera que cada elemento do resultado seja um vetor numérico de comprimento 1. Como esse NÃO é realmente o caso, VOCÊ OBTERÁ UM ERRO. Depois de obter o erro, digite ok () para passar para a próxima pergunta.

Como analista de dados, muitas vezes você deseja dividir seus dados em grupos com base no valor de alguma variável e, em seguida, aplicar uma função aos membros de cada grupo.

A próxima função que vamos olhar, tapply(), faz exatamente isso.

A variável "massa de terra" em nosso conjunto de dados assume valores inteiros entre 1 e 6, cada um dos quais representa uma parte diferente do mundo. Use tabela (bandeiras$massa terrestre) para ver quantas bandeiras/países possuem em cada grupo.

A variável 'animar' em nosso conjunto de dados leva o valor 1 se a bandeira de um país contiver uma imagem animada (por exemplo, uma águia, uma árvore, uma mão humana) e 0 de outra forma.

Use a tabela (bandeiras$animate) para ver quantas bandeiras contêm uma imagem animada.

Isso nos diz que 39 bandeiras contêm um objeto animado (anima = 1) e 155 não (anima = 0).

Se você tomar a média aritmética de um bando de 0s e 1s, você recebe a proporção de 1s. Use tapply (bandeiras$anima, bandeiras$massa terrestre, média) para aplicar a função média à variável 'animar' separadamente para cada um dos seis grupos de massa terrestre, dando-nos assim a proporção de bandeiras contendo uma imagem animada dentro de cada grupo de massa terrestre.

O primeiro grupo de massa terrestre (massa terrestre = 1) corresponde à América do Norte e contém a maior proporção de bandeiras com uma imagem animada (0,4194).

Da mesma forma, podemos ver um resumo dos valores da população (em milhões redondos) para países com e sem a cor vermelha em sua bandeira com tapply (sinalizadores $ população, sinalizadores $ vermelho, resumo).

Por último, use a mesma abordagem para examinar um resumo dos valores populacionais para cada uma das seis massas de terra.

Você pode ver um resumo das populações de cada uma das seis massas de terra chamando tapply () com três argumentos: flags $ população, flags $ landmass e summary.

Nesta lição, você aprendeu como usar vapply () como uma alternativa mais segura para sapply (), que é mais útil ao escrever suas próprias funções. Você também aprendeu como usar tapply () para dividir seus dados em grupos com base no valor de alguma variável e, em seguida, aplicar uma função a cada grupo. Essas funções serão úteis em sua busca para se tornar um analista de dados melhor.

**Semana 4 - swirl Lesson 1: Looking at Data**

Sempre que você estiver trabalhando com um novo conjunto de dados, a primeira coisa que você deve fazer é olhar

Nisso! Qual é o formato dos dados? Quais são as dimensões? O que são as nomes de variáveis? Como as variáveis ​​são armazenadas? Estão faltando dados? Existem falhas nos dados?

Esta lição irá ensiná-lo a responder a essas perguntas e muito mais usando R's funções integradas. Usaremos um conjunto de dados construído a partir do banco de dados PLANTS do Departamento de Agricultura dos Estados Unidos

(<http://plants.usda.gov/adv_search.html>).

Sempre que você estiver trabalhando com um novo conjunto de dados, a primeira coisa que você deve fazer é olhar Nisso! Qual é o formato dos dados? Quais são as dimensões? O que são as nomes de variáveis? Como as variáveis ​​são armazenadas? Estão faltando dados? Existem falhas nos dados?

Esta lição irá ensiná-lo a responder a essas perguntas e muito mais usando R's funções integradas. Estaremos usando um conjunto de dados construído a partir do banco de dados PLANTS do Departamento de Agricultura dos Estados Unidos

(http://plants.usda.gov/adv\_search.html).

Eu armazenei os dados para você em uma variável chamada plantas. Digite ls () para listar as variáveis ​​em sua área de trabalho, entre as quais devem estar as plantas.

Vamos começar verificando a classe da variável plants com class (plants). Isso nos dará uma pista sobre a estrutura geral dos dados.

É muito comum que os dados sejam armazenados em um quadro de dados. É a classe padrão para dados lidos em R usando funções como read.csv () e read.table (), que você aprenderá em outra lição.

Como o conjunto de dados é armazenado em um quadro de dados, sabemos que é retangular. Em outras palavras, ele tem duas dimensões (linhas e colunas) e se encaixa perfeitamente em uma tabela ou planilha. Use dim (plantas) para ver exatamente quantas linhas e colunas estamos lidar com.

O primeiro número que você vê (5166) é o número de linhas (observações) e o segundo número (10) é o número de colunas (variáveis).

Você também pode usar nrow (plantas) para ver apenas o número de linhas. Experimente.

E ncol (plantas) para ver apenas o número de colunas.

Se você estiver curioso para saber quanto espaço o conjunto de dados está ocupando na memória, você pode usar object.size (plants).

Agora que temos uma noção da forma e do tamanho do conjunto de dados, vamos ter uma ideia do que está dentro. nomes (plantas) retornarão um vetor de caracteres de coluna (ou seja, nomes de variáveis). Dê uma chance.

Aplicamos nomes de variáveis ​​bastante descritivos a este conjunto de dados, mas isso não sempre será o caso. Uma próxima etapa lógica é examinar os dados reais. No entanto, nosso conjunto de dados contém mais de 5.000 observações (linhas), por isso é impraticável visualizar tudo de uma vez.

A função head () permite que você visualize o topo do conjunto de dados. Experimente com apenas um argumento.

Reserve um minuto para examinar e compreender o resultado acima. Cada linha é rotulada com o número da observação e cada coluna com o nome da variável. Sua tela provavelmente não é larga o suficiente para ver todas as 10 colunas lado a lado, nas quais o caso R exibe quantas colunas puder em cada linha antes de continuar na próxima.

Por padrão, head () mostra as primeiras seis linhas dos dados. Você pode alterar isso comportamento passando como um segundo argumento o número de linhas que você gostaria de ver. Use head () para visualizar as primeiras 10 fileiras de plantas.

O mesmo se aplica ao uso de tail () para visualizar o final do conjunto de dados. Use tail () para ver as últimas 15 linhas.

Depois de visualizar as partes superior e inferior dos dados, você provavelmente notou muitos NAs, que são marcadores de posição de R para valores ausentes. Use o resumo (plantas) para ter uma ideia melhor de como cada variável é distribuída e quanto do conjunto de dados está faltando.

summary () fornece saída diferente para cada variável, dependendo de sua classe.

Para dados numéricos como Precip\_Min, summary () exibe o mínimo, 1º quartil, mediana, média, 3º quartil e máximo. Esses valores nos ajudam a entender como os dados são distribuídos.

Para variáveis ​​categóricas (chamadas de variáveis ​​de 'fator' em R), o resumo () exibe o número de vezes que cada valor (ou 'nível') ocorre nos dados. Por exemplo, cada valor de Scientific\_Name aparece apenas uma vez, uma vez que é exclusivo para uma planta específica. Em contraste, o resumo da duração (também uma variável de fator) nos diz que nosso conjunto de dados contém 3031 plantas perenes, 682 plantas anuais, etc.

Você pode ver que R truncou o resumo de Active\_Growth\_Period ao incluir uma categoria abrangente chamada 'Outro'. Uma vez que é uma variável categórica / fator, podemos ver quantas vezes cada valor realmente ocorre nos dados com

tabela (plantas $ Active\_Growth\_Period).

Cada uma das funções que apresentamos até agora serve para ajudá-lo a compreender melhor a estrutura dos seus dados. No entanto, deixamos o melhor para durar....

Talvez a função mais útil e concisa para entender a \* estrutura \* de seus dados seja str (). Experimente agora.

A beleza de str () é que ele combina muitos dos recursos de outras funções que você já viu, tudo em um formato conciso e legível. No topo, ele nos diz que a classe de plantas é 'data.frame' e que tem 5166 observações e 10 variáveis. Em seguida, ele nos fornece o nome e a classe de cada variável, bem como uma visualização de seu conteúdo.

str () é na verdade uma função muito geral que você pode usar na maioria dos objetos em R.

Sempre que você quiser entender a estrutura de algo (um conjunto de dados, função, etc.), str () é um bom lugar para começar.

Nesta lição, você aprendeu como ter uma ideia da estrutura e do conteúdo de um novo conjunto de dados usando uma coleção de funções simples e úteis. Reservar um tempo para fazer isso antecipadamente pode economizar tempo e frustração mais tarde em sua análise.

**### Semana 4 - swrl Lesson 2: Simulation**

Uma das grandes vantagens de usar uma linguagem de programação estatística como R é sua vasta coleção de ferramentas para simular números aleatórios.

Esta lição pressupõe familiaridade com algumas distribuições de probabilidade comuns, mas esses tópicos serão discutidos apenas em relação à geração de números aleatórios.

Mesmo que você não tenha experiência anterior com esses conceitos, deverá ser capaz de concluir a lição e compreender as idéias principais.

A primeira função que usaremos para gerar números aleatórios é sample (). Use? Sample para obter a documentação.

Vamos simular o lançamento de quatro dados de seis lados: amostra (1: 6, 4, substituir = VERDADEIRO).

Agora repita o comando para ver como o seu resultado difere. (A probabilidade de obter exatamente o mesmo resultado é (1/6) ^ 4 = 0,00077, o que é muito pequeno!)

amostra (1: 6, 4, substituir = VERDADEIRO) instrui R a selecionar aleatoriamente quatro números entre 1 e 6, COM substituição. Amostragem com substituição significa simplesmente que cada número é "substituído" após ser selecionado, de forma que o mesmo número possa aparecer mais de uma vez. Isso é o que queremos aqui, já que o que você rola em um dado não deve afetar o que você rola em nenhum dos outros.

Agora experimente 10 números entre 1 e 20, SEM substituição. Para provar sem

substituição, simplesmente deixe de fora o argumento 'substituir'.

LETTERS é uma variável predefinida em R contendo um vetor de todas as 26 letras do alfabeto inglês. Dê uma olhada nisso agora.

A função sample () também pode ser usada para permutar ou reorganizar os elementos de um vetor. Por exemplo, tente sample (LETTERS) para permutar todas as 26 letras do

Alfabeto Ingles.

Isso é idêntico a tirar uma amostra de tamanho 26 de LETRAS, sem reposição. Quando o argumento 'size' para sample () não é especificado, R pega uma amostra igual em tamanho ao vetor do qual você está amostrando.

Agora, suponha que queremos simular 100 lançamentos de uma moeda injusta de dois lados. Esta determinada moeda tem uma probabilidade de 0,3 de cair 'coroa' e uma probabilidade de 0,7 de cair 'cara'.

Deixe o valor 0 representar coroa e o valor 1 representar cara. Use sample () para extrair uma amostra de tamanho 100 do vetor c (0,1), com substituição. Uma vez que a moeda é injusta, devemos atribuir probabilidades específicas aos valores 0 (coroa) e

1 (cara) com um quarto argumento, prob = c (0,3, 0,7). Atribua o resultado a uma nova variável chamada flips.

vira <- amostra (c (0,1), 100, substituir = VERDADEIRO, prob = c (0,3, 0,7))

Uma vez que definimos a probabilidade de aterrissar cabeças em qualquer flip dado como 0,7, nós esperamos que aproximadamente 70 de nossos lançamentos de moeda tenham o valor 1. Conte o número real de 1s contido nos lançamentos usando a função sum ().

sum (flips) irá somar todos os 1s e 0s, dando-lhe o número total de 1s em vira.

Um cara ou coroa é um resultado binário (0 ou 1) e estamos realizando 100 independentes

tentativas (cara ou coroa), então podemos usar rbinom () para simular um binômio aleatório

variável. Puxe a documentação para rbinom () usando? Rbinom.

Cada distribuição de probabilidade em R tem uma função r \*\*\* (para "aleatório"), função ad \*\*\* (para "densidade"), ap \*\*\* (para "probabilidade") e q \*\*\* (para " quantil ").

Estamos mais interessados ​​nas funções r \*\*\* nesta lição, mas eu encorajo você

para explorar os outros por conta própria.

Uma variável aleatória binomial representa o número de 'sucessos' (cara) em um determinado número de 'tentativas' independentes (cara ou coroa). Portanto, podemos gerar um

única variável aleatória que representa o número de caras em 100 lançamentos de nossa moeda injusta usando rbinom (1, tamanho = 100, prob = 0,7). Observe que você só especifica

a probabilidade de 'sucesso' (cara) e NÃO a probabilidade de 'falha' (coroa). Tente agora.

Uma variável aleatória binomial representa o número de 'sucessos' (cara) em um determinado número de 'tentativas' independentes (cara ou coroa). Portanto, podemos gerar um

única variável aleatória que representa o número de caras em 100 lançamentos de nossa moeda injusta usando rbinom (1, tamanho = 100, prob = 0,7). Observe que você só especifica

a probabilidade de 'sucesso' (cara) e NÃO a probabilidade de 'falha' (coroa). Tente agora.

Da mesma forma, se quisermos ver todos os 0s e 1s, podemos solicitar 100 observações, cada uma de tamanho 1, com probabilidade de sucesso de 0,7. De uma chance,

atribuindo o resultado a uma nova variável chamada flips2.

Chame rbinom () com n = 100, tamanho = 1 e prob = 0,7 e atribua o resultado a

flips2.

Agora use sum () para contar o número de 1s (cara) em flips2. Deve estar perto de 70!

Semelhante a rbinom (), podemos usar R para simular números aleatórios de muitos outros

distribuições de probabilidade. Puxe a documentação para rnorm () agora.

A distribuição normal padrão tem média 0 e desvio padrão 1. Como você pode ver na seção 'Uso' na documentação, os valores padrão para o

Os argumentos 'mean' e 'sd' para rnorm () são 0 e 1, respectivamente. Assim, rnorm (10)

irá gerar 10 números aleatórios de uma distribuição normal padrão. De uma chance.

Agora faça o mesmo, exceto com uma média de 100 e um desvio padrão de 25.

Finalmente, e se quisermos simular 100 \* grupos \* de números aleatórios, cada

contendo 5 valores gerados a partir de uma distribuição de Poisson com média 10? Vamos

comece com um grupo de 5 números, então mostrarei como repetir a operação 100 vezes de uma maneira conveniente e compacta.

Gere 5 valores aleatórios de uma distribuição de Poisson com média 10. Verifique a documentação de rpois () se precisar de ajuda.

Agora use replicate (100, rpois (5, 10)) para realizar esta operação 100 vezes. Loja

o resultado em uma nova variável chamada my\_pois.

replicate () criou uma matriz, cada coluna contendo 5 números aleatórios

gerado a partir de uma distribuição de Poisson com média 10. Agora podemos encontrar a média de cada coluna em my\_pois usando a função colMeans (). Armazene o resultado em um

variável chamada cm.

E vamos dar uma olhada na distribuição de nossos meios de coluna traçando um

histograma com hist (cm).

Parece que as médias das nossas colunas são distribuídas quase normalmente, certo? Esse é o Teorema do Limite Central em ação, mas essa é uma lição para outro dia!

Todas as distribuições de probabilidade padrão são construídas em R, incluindo exponencial (rexp ()), qui-quadrado (rchisq ()), gama (rgamma ()), .... Bem, você

veja o padrão.

A simulação é praticamente um campo próprio e nós apenas examinamos a superfície do que é possível. Eu encorajo você a explorar ainda mais essas e outras funções

por si só.

**swirl Lesson 3: Base Graphics**

Um dos maiores pontos fortes de R, em relação a outras linguagens de programação, é a facilidade com que podemos criar gráficos de qualidade de publicação. Nesta lição, você aprenderá sobre gráficos básicos em R.

Nós não cobrimos as porções mais avançadas de gráficos em R nesta lição. Estes incluem lattice, ggplot2 e ggvis.

Há uma escola de pensamento de que essa abordagem é ao contrário, que devemos ensinar ggplot2 primeiro. Ver http://varianceexplained.org/r/teach\_ggplot2\_to\_beginners/ para um esboço dessa visão.

Carregue o dataframe cars com data(cars).

Para corrigir ideias, trabalharemos com dataframe de dados simples. Nosso principal objetivo é apresentar várias funções de plotagem e seus argumentos. Toda a saída pareceria mais

interessante com conjuntos de dados maiores e mais complexos.

Abra a página de ajuda para carros.

?cars

Como você pode ver na página de ajuda, o conjunto de dados dos carros tem apenas duas variáveis: velocidade e distância de parada. Observe que os dados são da década de 1920.

Execute head () nos dados dos carros.

Antes de plotar, é sempre uma boa ideia ter uma noção dos dados. Chave R

os comandos para fazer isso incluem dim (), names (), head (), tail () e summary ().

Execute o comando plot () no quadro de dados do carro.

para criar um gráfico do quadro de dados dos carros.

Como sempre, R tenta muito dar a você algo sensato, dada a informação que você forneceu a ele. Primeiro, R observa que o quadro de dados que você forneceu tem apenas duas colunas, então ele assume que você deseja plotar uma coluna em relação à outra.

Agora, passaremos algum tempo explorando o gráfico, mas muitos dos tópicos cobertos aqui se aplicarão à maioria das outras funções gráficas R. Observe que 'plot' é a abreviação de gráfico de dispersão.

Procure na página de ajuda plot ().

A página de ajuda para plot () destaca os diferentes argumentos que a função pode assumir. Os dois mais importantes são x e y, as variáveis ​​que serão plotadas. Para que no próximo conjunto de perguntas, inclua os nomes dos argumentos em suas respostas. Ou seja, não digite plot (cars $ speed, cars $ dist), embora funcione. Em vez disso, use plot (x = cars $ speed, y = cars $ dist).

Use o comando plot () para mostrar a velocidade no eixo x e dist no eixo y a partir do quadro de dados dos carros. Use a forma do comando plot no qual os vetores são explicitamente passados ​​como argumentos para x e y.

Observe que isso produz uma resposta ligeiramente diferente do gráfico (carros). Neste caso, R não tem certeza do que você deseja usar como rótulos nos eixos, então ele apenas

usa os argumentos que você passa, nome do quadro de dados e cifrões incluídos.

Observe que existem outras maneiras de chamar o comando plot, ou seja, usando o

interface de "fórmula". Por exemplo, obtemos um gráfico semelhante ao anterior com o gráfico (dist ~ velocidade, carros). No entanto, vamos esperar até mais tarde na lição antes

usando a interface de fórmula.

Use o comando plot () para mostrar dist no eixo x e speed no eixo y a partir do dataframe cars. Isso é o oposto do que fizemos acima.

plot (x = cars $ dist, y = cars $ speed).

Provavelmente faz mais sentido a velocidade ir no eixo x, uma vez que a distância de parada é uma função da velocidade mais do que o contrário. Portanto, para o restante das perguntas nesta parte da lição, sempre atribua os argumentos de acordo.

Na verdade, você pode assumir que as respostas para as próximas perguntas são todas do gráfico de forma (x = carros $ velocidade, y = carros $ dist, ...), mas com vários argumentos usados ​​no lugar do ...

Recrie o gráfico com o rótulo do eixo x definido como "Velocidade"

plot(x = cars$speed, y = cars$dist, xlab = "Speed")

Recrie o gráfico com "Velocidade" e "Distância de parada" como rótulos de eixo.

plot(x = cars$speed, y = cars$dist, xlab = "Speed", ylab = "Stopping Distance")

A razão pela qual os gráficos (carros) funcionaram no início da lição foi que R era inteligente o suficiente para saber que o primeiro elemento (ou seja, a primeira coluna) nos carros deve ser atribuído ao argumento xe o segundo elemento ao argumento y . Para economize na digitação, o próximo conjunto de respostas será todo na forma, enredo (carros,...) com vários argumentos adicionados.

Para cada pergunta, queremos apenas um argumento adicional de cada vez. Claro, você pode transmitir mais de um argumento ao fazer um projeto real.

Plote carros com o título principal de "Meu Plot". Observe que o argumento para o título principal é "principal" e não "título".

plot(cars, main = "My Plot")

A página de ajuda de plot (? Plot) cobre apenas um pequeno número dos muitos argumentos que podem ser passados ​​para plot () e outras funções gráficas. Para começar a explorar as muitas outras opções, olhe para? Par. Vejamos alguns dos mais comumente usados. Continue usando plot (carros, ...) como a resposta básica para essas perguntas.

Trace os carros de forma que os pontos traçados sejam coloridos de vermelho. (Use col = 2 para obter este efeito.)

plot(cars, col = 2)

Plote os carros enquanto limita o eixo x de 10 a 15. (Use xlim = c (10, 15) para obter este efeito.)

Você também pode alterar a forma dos símbolos no gráfico. A página de ajuda para pontos (?points) fornece os detalhes.

plot(cars, pch = 2)

Argumentos como "col" e "pch" podem não parecer muito intuitivos. E isso é porque eles não são! Portanto, muitas / muitas pessoas usam pacotes mais modernos, como ggplot2, para

criando seus gráficos em R.

É, no entanto, útil ter uma introdução aos gráficos básicos porque muitos dos

os idiomas em lattice e ggplot2 são modelados neles.

Vejamos agora algumas outras funções em gráficos básicos que podem ser úteis, começando com boxplots.

Carregar o data frame mtcars

data(mtcars)

Sempre que você carregar um novo quadro de dados, deverá explorá-lo antes de usá-lo.

No meio de uma lição de turbulência, basta digitar play (). Isso suspende temporariamente a lição (sem perder o trabalho que você já fez) e permite que você emita comandos como dim (mtcars) e head (mtcars). Quando terminar de examinar os dados, basta digitar nxt () e a lição continuará de onde parou.

Procure na página de ajuda boxplot ().

Em vez de adicionar colunas de dados diretamente como argumentos de entrada, como fizemos com

plot (), geralmente é útil passar todo o quadro de dados. Isso é o que

O argumento "data" em boxplot () permite.

boxplot (), como muitas funções R, também leva um argumento de "fórmula", geralmente um

expressão com um til ("~") que indica a relação entre as variáveis ​​de entrada. Isso permite que você insira algo como mpg ~ cyl para plotar a relação entre cyl (número de cilindros) no eixo x e mpg (milhas por galão) no eixo y.

Use boxplot () com formula = mpg ~ cyl e data = mtcars para criar um gráfico de caixa.

boxplot(formula = mpg ~cyl, data = mtcars)

O gráfico mostra que o mpg é muito menor para carros com mais cilindros. Observe que podemos usar o mesmo conjunto de argumentos que exploramos com plot () acima para adicionar rótulos de eixo, títulos e assim por diante.

Ao olhar para uma única variável, os histogramas são uma ferramenta útil. hist () é a função R associada. Como plot (), hist () é melhor usado apenas passando um

único vetor.

hist(mtcars$mpg) ## para criar histograma

Nesta lição, você aprendeu como trabalhar com gráficos básicos em R. O melhor lugar para ir a partir daqui é estudar o pacote ggplot2. Se você quiser explorar outros elementos dos gráficos básicos, esta página da web (http://www.ling.upenn.edu/~joseff/rstudy/week4.html) fornece uma visão geral útil.