YOU AND YOUR FRIENDS ARE PLAYING A GAME OF MONOPOLY

EXAMPLE 6: CREATING A VECTOR

LET'S CREATE A VECTOR TO KEEP TRACK OF HOW MUCH MONEY EACH PLAYER HAS LEFT

A VECTOR IS A COLLECTION OF ELEMENTS, ALL OF THE SAME TYPE

EXAMPLE 6: CREATING A VECTOR

A VECTOR IS A COLLECTION OF ELEMENTS, ALL OF THE SAME TYPE

```
playerMoney <- c(247, 350, 130, 4000, 600)
playerMoney
```

```
[1]  247  350  130 4000  600
```

THE C() FUNCTION IS THE MOST POPULAR WAY TO SET UP A VECTOR

EXAMPLE 6: CREATING A VECTOR

C STANDS FOR COMBINE (OR) CONCATENATE

```
playerMoney <- c(247, 350, 130, 4000, 600)
playerMoney
```

```
[1]  247  350  130 4000  600
```

IT CAN TAKE ANY NUMBER OF INPUTS AND COMBINE THEM INTO A VECTOR

EXAMPLE 7: THE MODE OF A VECTOR

EXAMPLE 7: THE MODE OF A VECTOR

```
playerMoney <- c(247, 350, 130, 4000, 600)
playerMoney
```

```
[1]  247  350  130 4000  600
```

```
mode(playerMoney)
```

```
[1] "numeric"
```

THE TYPE OF THE ELEMENTS OF A VECTOR IS CALLED IT'S MODE

EXAMPLE 7: THE MODE OF A VECTOR

```
playerMoney <- c(247, 350, 130, 4000, 600)
playerMoney
```

```
[1]  247  350  130 4000  600
```

```
mode(playerMoney)
```

```
[1] "numeric"
```

THE **MODE** OF A VECTOR CAN BE ONE OF ONLY **5 BASIC TYPES**

THE MODE OF A VECTOR CAN BE ONE OF ONLY 5 BASIC TYPES

NUMBERS (BOTH INTEGERS AND FLOATS)	NUMERIC	(1, 3.3, 5, 7)
STRINGS	CHARACTER	("A", "VECTOR", "OF", "CHARACTERS")
CAN ONLY TAKE VALUES TRUE OR FALSE	LOGICAL	(FALSE, TRUE, FALSE, FALSE)
COMPLEX NUMBERS	COMPLEX	(3+2i, 4.5+0i, 10+32i)
BYTES	RAW	(0A, 1E, 28, 3C)

EXAMPLE 7: THE MODE OF A VECTOR

```
playerMoney <- c(247, 350, 130, 4000, 600)
playerMoney
```

```
[1]  247  350  130 4000  600
```

```
mode(playerMoney)
```

```
[1] "numeric"
```

```
startPlayerMoney <- numeric(5)
```

YOU CAN ALSO CREATE A VECTOR USING THE MODE AND LENGTH OF THE VECTOR

EXAMPLE 7: THE MODE OF A VECTOR

```
playerMoney <- c(247, 350, 130, 4000, 600)
playerMoney
```

```
[1]  247  350  130 4000  600
```

```
mode(playerMoney)
```

```
[1] "numeric"
```

```
startPlayerMoney <- numeric(5)
startPlayerMoney
```

```
[1] 0 0 0 0 0
```

THE DEFAULT VALUE DEPENDS ON THE MODE

THE RESULT WILL BE A VECTOR WITH ALL ELEMENTS EQUAL TO A DEFAULT VALUE

EXAMPLE 7: THE MODE OF A VECTOR

```
playerNames <- c(1, 2, "Dave", "Gru", "Edith")
```

IF YOU TRY TO CREATE A VECTOR WITH ELEMENTS OF DIFFERENT TYPES

EXAMPLE 7: THE MODE OF A VECTOR

CONVERTED TO CHARACTER

IF YOU TRY TO CREATE A VECTOR WITH ELEMENTS OF DIFFERENT TYPES

```
playerNames <- c(1, 2, "Dave", "Gru", "Edith")
playerNames
[1] "1"     "2"     "Dave"  "Gru"   "Edith"
```

ALL THE ELEMENTS WILL BE FORCED TO A SINGLE TYPE

THE TYPE CHOSEN IS ONE TO WHICH ALL THE ELEMENTS CAN BE CONVERTED

EXAMPLE 8: VECTORS ARE ATOMIC

EXAMPLE 8: VECTORS ARE ATOMIC

"VECTOR" IS ACTUALLY SHORT FOR

ATOMIC VECTOR

EXAMPLE 8: VECTORS ARE ATOMIC

EACH ELEMENT IN A VECTOR IS
"ATOMIC" I.E. IT'S A SINGLE ELEMENT

IN OTHER WORDS, A VECTOR CANNOT
CONSIST OF OTHER VECTORS

WHAT HAPPENS IF YOU TRY TO CREATE A VECTOR WHICH CONSISTS OF OTHER VECTORS?

EXAMPLE 8: VECTORS ARE ATOMIC

```
playerMoneyTrend <- c(startPlayerMoney, playerMoney)
```

THE VECTORS WILL BE CONCATENATED INTO A SINGLE VECTOR

EXAMPLE 8: VECTORS ARE ATOMIC

```
playerMoney <- c(247, 350, 130, 4000, 600)
playerMoney
[1]  247  350  130 4000  600
mode(playerMoney)
[1] "numeric"
startPlayerMoney <- numeric(5)
startPlayerMoney
[1] 0 0 0 0 0
playerNames <- c(1, 2, "Dave", "Gru", "Edith")
playerNames
[1] "1"     "2"     "Dave"  "Gru"   "Edith"
playerMoneyTrend <- c(startPlayerMoney, playerMoney)
```

THE VECTORS WILL BE CONCATENATED INTO A SINGLE VECTOR

EXAMPLE 8: VECTORS ARE ATOMIC

```
playerMoney <- c(247, 350, 130, 4000, 600)
playerMoney
[1]  247  350  130 4000  600
mode(playerMoney)
[1] "numeric"
startPlayerMoney <- numeric(5)
startPlayerMoney
[1] 0 0 0 0 0
playerNames <- c(1, 2, "Dave", "Gru", "Edith")
playerNames
[1] "1"     "2"     "Dave"  "Gru"   "Edith"
playerMoneyTrend <- c(startPlayerMoney, playerMoney)
```

THE VECTORS WILL BE CONCATENATED INTO A SINGLE VECTOR

0 0 0 0 0

EXAMPLE 8: VECTORS ARE ATOMIC

```
playerMoney <- c(247, 350, 130, 4000, 600)
playerMoney
[1]  247  350  130 4000  600
mode(playerMoney)
[1] "numeric"
startPlayerMoney <- numeric(5)
startPlayerMoney
[1] 0 0 0 0 0
playerNames <- c(1, 2, "Dave", "Gru", "Edith")
playerNames
[1] "1"     "2"     "Dave"  "Gru"   "Edith"
playerMoneyTrend <- c(startPlayerMoney, playerMoney)
playerMoneyTrend
[1]    0    0    0    0    0  247  350  130 4000  600
```

THE VECTORS WILL BE CONCATENATED INTO A SINGLE VECTOR

0 0 0 0 0 247 350 130 4000 600

EXAMPLE 8: VECTORS ARE ATOMIC

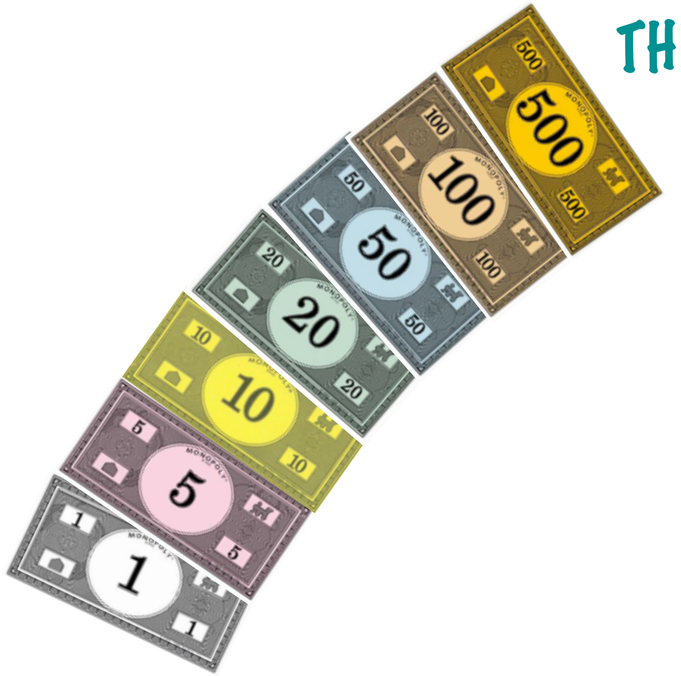
```
playerMoney <- c(247, 350, 130, 4000, 600)
playerMoney
[1]  247  350  130 4000  600
mode(playerMoney)
[1] "numeric"
startPlayerMoney <- numeric(5)
startPlayerMoney
[1] 0 0 0 0 0
playerNames <- c(1, 2, "Dave", "Gru", "Edith")
playerNames
[1] "1"     "2"     "Dave"  "Gru"   "Edith"
playerMoneyTrend <- c(startPlayerMoney, playerMoney)
playerMoneyTrend
[1]    0    0    0    0    0  247  350  130 4000  600
```

LET'S GET BACK TO OUR MONOPOLY GAME

EXAMPLE 9: DO SOMETHING TO EACH ELEMENT OF A VECTOR

EXAMPLE 9: DO SOMETHING TO EACH ELEMENT OF A VECTOR

1) SET UP A BANK



THERE ARE 7 DENOMINATIONS

```
denominations <- c(1, 5, 10, 20, 50, 100, 500)
```

THERE ARE 30 BILLS PER DENOMINATION

HOW WOULD YOU FIND THE AMOUNT PER DENOMINATION?

EXAMPLE 9: ADDING A NUMBER TO ALL ELEMENTS OF A VECTOR

```
amountPerDenomination <- denominations * 30
amountPerDenomination
[1]    30   150   300   600  1500  3000 15000
```

ANY OPERATION ON A VECTOR, BY ANOTHER NUMBER, WILL BE APPLIED TO EACH ELEMENT OF THE VECTOR

EXAMPLE 9: DO SOMETHING TO EACH ELEMENT OF A VECTOR

```
denominations <- c(1, 5, 10, 20, 50, 100, 500)
amountPerDenomination <- denominations * 30
amountPerDenomination
[1]    30   150   300   600  1500  3000 15000
```

THIS IS LIKE **MAGIC**

IN MOST OTHER PROGRAMMING LANGUAGES

THE ONLY WAY TO DO THIS IS TO WRITE A LOOP TO MULTIPLY EACH ELEMENT OF THE VECTOR

```
amountPerDen <- numeric()
for(i in denominations) {
amountPerDen <- c(amountPerDen , i*30)
}
```

THIS MIND-BENDING BIT OF CODE - CAN BE SAFELY REPLACED WITH SOMETHING COMPACT AND BEAUTIFUL

EXAMPLE 9: DO SOMETHING TO EACH ELEMENT OF A VECTOR

ANY OPERATION ON A VECTOR, BY ANOTHER NUMBER, WILL BE APPLIED TO EACH ELEMENT OF THE VECTOR

\+ - / * ^

THESE OPERATIONS WILL BE APPLIED ELEMENT-WISE TO THE VECTOR

FUNCTIONS LIKE LOG(), SIN(), SQRT() ETC WILL ALSO BE APPLIED TO EACH ELEMENT OF THE VECTOR

EXAMPLE 10: AGGREGATING ALL ELEMENTS OF A VECTOR

EXAMPLE 10: AGGREGATING ALL ELEMENTS OF A VECTOR

```
totalAmountInBank <- sum(amountPerDenomination)
```

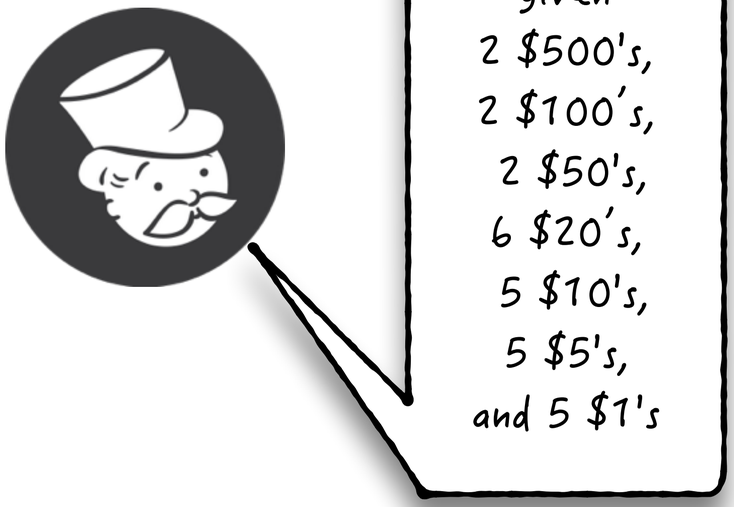
SUM() WILL GIVE THE TOTAL OF ALL THE ELEMENTS IN A VECTOR

PRODUCT() WILL GIVE THE PRODUCT OF ALL THE ELEMENTS IN A VECTOR

MEAN() WILL GIVE THE AVERAGE OF ALL THE ELEMENTS IN A VECTOR

EXAMPLE 1 1: OPERATIONS BETWEEN VECTORS OF SAME LENGTH

EXAMPLE 11: OPERATIONS BETWEEN VECTORS OF SAME LENGTH



$1's $10's $50's $500's

```
startMoney <- c(5, 5, 5, 6, 2, 2, 2)
```

$5's $20's $100's

HOW WOULD YOU FIND THE AMOUNT PER DENOMINATION?

```
startAmountPerDen <- startMoney * denominations
```

ANY OPERATION BETWEEN VECTORS OF THE SAME LENGTH IS APPLIED ELEMENT-WISE

EXAMPLE 11: OPERATIONS BETWEEN VECTORS OF SAME LENGTH

```
startMoney <- c(5, 5, 5, 6, 2, 2, 2)
startAmountPerDen <- startMoney * denominations
startAmountPerDen
```

```
[1]    5   25   50  120  100  200 1000
```

DENOMINATIONS	(1,	5,	10,	20,	50,	100,	500)
START MONEY	(5,	5,	5,	6,	2,	2,	2)

ANY OPERATION BETWEEN VECTORS OF THE SAME LENGTH IS APPLIED ELEMENT-WISE

THIS WORKS WITH ANY OPERATION +, -, /, ^, * OR EVEN LOGICAL OPERATORS >, <, >= ETC

EXAMPLE 11: OPERATIONS BETWEEN VECTORS OF SAME LENGTH

LET'S SEE ANOTHER EXAMPLE

```
startMoney <- c(5, 5, 5, 6, 2, 2, 2)
startAmountPerDen <- startMoney * denominations
startAmountPerDen
[1]    5  25  50 120 100 200 1000
```

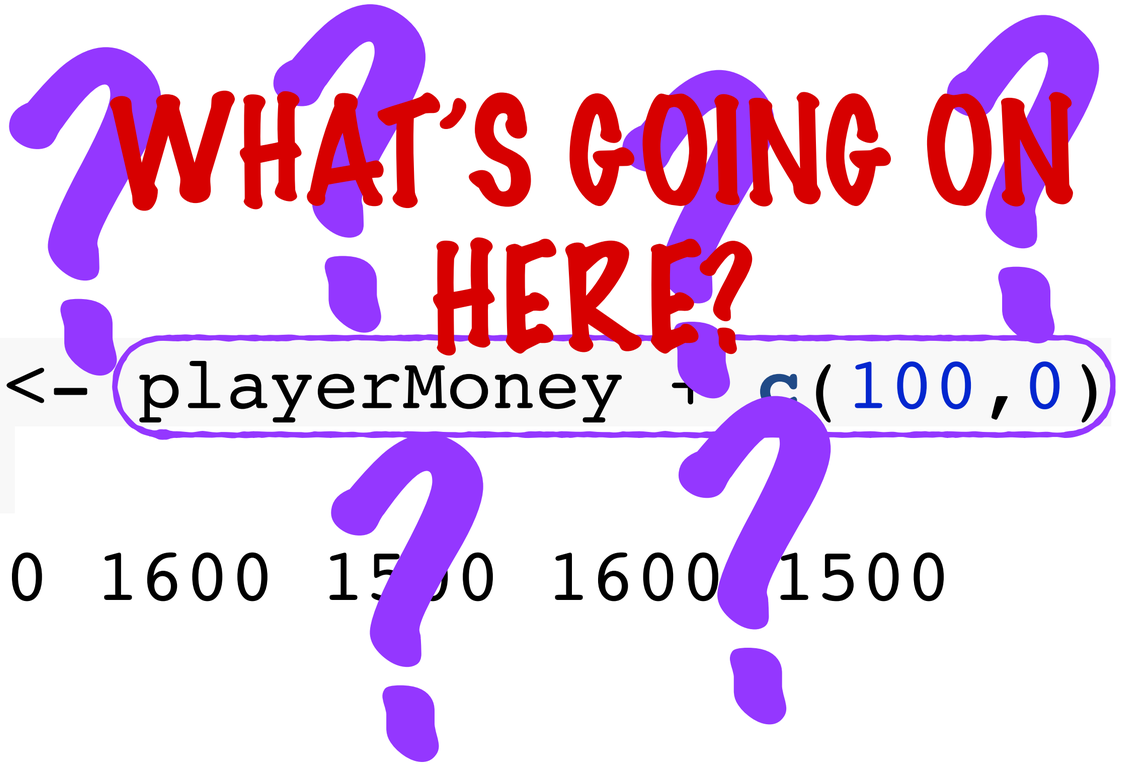
```
playerMoney <- numeric(6) + sum(startAmountPerDen
```

(0, 0, 0, 0, 0, 0) + 1500

```
playerMoney
[1] 1500 1500 1500 1500 1500 1500
```

EXAMPLE 12: OPERATIONS BETWEEN VECTORS OF DIFFERENT LENGTHS

EXAMPLE 12: OPERATIONS BETWEEN VECTORS OF DIFFERENT LENGTHS



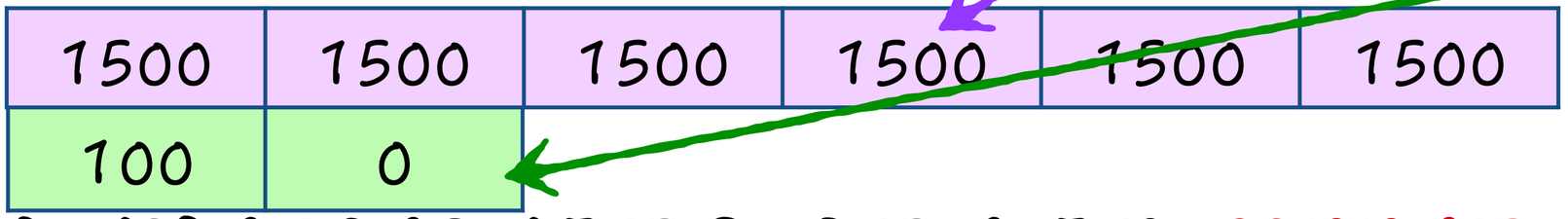

```
playerMoney <- playerMoney + c(100,0)
playerMoney
```

```
[1] 1600 1500 1600 1500 1600 1500
```

EXAMPLE 12: OPERATIONS BETWEEN VECTORS OF DIFFERENT LENGTHS

```
playerMoney
[1] 1500 1500 1500 1500 1500 1500
```

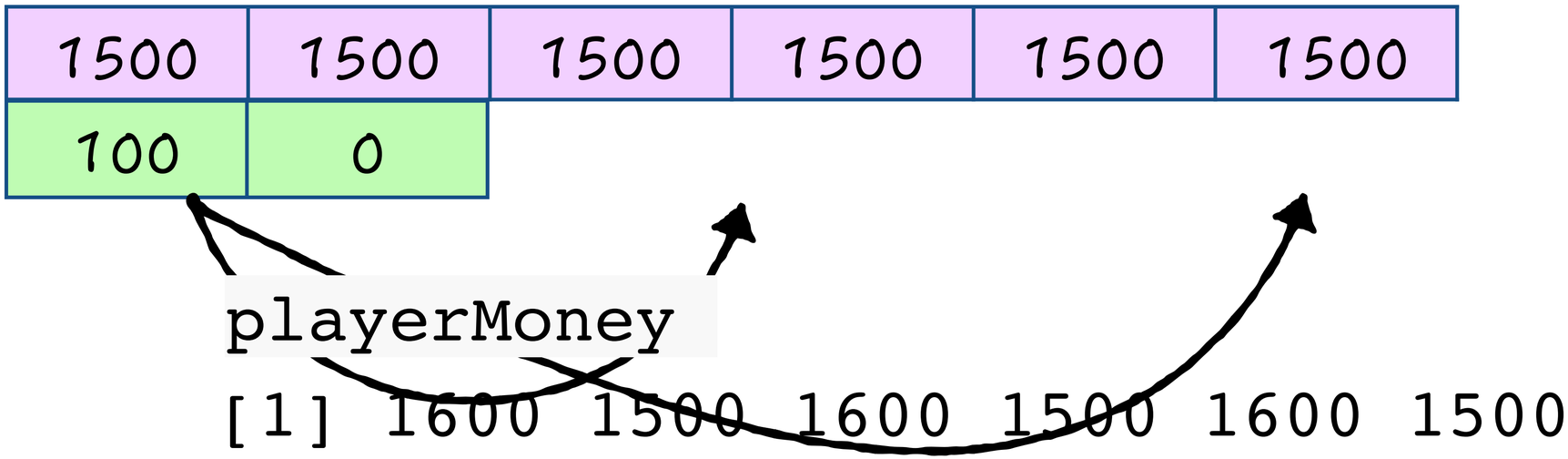
```
playerMoney <- playerMoney + c(100,0)
```



WHEN THE VECTORS ARE OF DIFFERENT LENGTHS, THE SHORTER VECTOR IS RE-USED AS MANY TIMES AS NEEDED

EXAMPLE 12: OPERATIONS BETWEEN VECTORS OF DIFFERENT LENGTHS

```
playerMoney <- playerMoney + c(100,0)
```



WHEN THE VECTORS ARE OF DIFFERENT LENGTHS, THE SHORTER VECTOR IS RE-USED AS MANY TIMES AS NEEDED

EXAMPLE 12: OPERATIONS BETWEEN VECTORS OF DIFFERENT LENGTHS

```
playerMoney <- playerMoney + c(100,0)
playerMoney
[1] 1600 1500 1600 1500 1600 1500
```

WHEN THE VECTORS ARE OF DIFFERENT LENGTHS, THE SHORTER VECTOR IS RE-USED AS MANY TIMES AS NEEDED

THIS IS KNOWN AS

RECYCLING

EXAMPLE 12: OPERATIONS BETWEEN VECTORS OF DIFFERENT LENGTHS

```
denominations <- c(1, 5, 10, 20, 50, 100, 500)
amountPerDenomination <- denominations * 30
amountPerDenomination
[1]    30   150   300   600  1500  3000 15000
```

WHEN YOU MULTIPLY A NUMBER WITH A VECTOR OF LENGTH 7

THE NUMBER IS RECYCLED 7 TIMES

THE NUMBER IS A VECTOR OF LENGTH 1

EXAMPLE 12: OPERATIONS BETWEEN VECTORS OF DIFFERENT LENGTHS

```
startMoney <- c(5, 5, 5, 6, 2, 2, 2)
startAmountPerDen <- startMoney * denominations
startAmountPerDen
[1]    5   25   50  120  100  200 1000
```

WHEN YOU MULTIPLY 2 VECTORS OF EQUAL LENGTH

EACH VECTOR IS USED EXACTLY ONCE

EXAMPLE 12: OPERATIONS BETWEEN VECTORS OF DIFFERENT LENGTHS

```
playerMoney <- playerMoney + c(100,0)
playerMoney
[1] 1600 1500 1600 1500 1600 1500
```

WHEN YOU ADD A VECTOR OF LENGTH 6 TO A VECTOR OF LENGTH 2

THE SHORTER VECTOR IS RECYCLED 3 TIMES

EXAMPLE 12: OPERATIONS BETWEEN VECTORS OF DIFFERENT LENGTHS

WHEN YOU ADD A VECTOR OF LENGTH 10 TO A VECTOR OF LENGTH 4

THE SHORTER VECTOR IS RECYCLED 2.5 TIMES

1	2	3	4	5	6	7	8	9	10
1	2	3	4	1	2	3	4	1	2

1 1 0.5

EXAMPLE 12: OPERATIONS BETWEEN VECTORS OF DIFFERENT LENGTHS

RECYCLING MAKES
PROGRAMMING IN R
COMPACT AND BEAUTIFUL

1 2 3 4 5 6 7 8 9 10

2 4 6 8 10

10 9 8 7 6 5 4 3 2 1

EXAMPLE 13: GENERATING SEQUENCES

2 2 2 4 4 4 6 6 6 8 8 8

-5.0 -4.8 -4.6 -4.4 -4.2 -4.0 -3.8

EXAMPLE 13: GENERATING SEQUENCES

1 2 3 4 5 6 7 8 9 10

2 4 6 8 10

10 9 8 7 6 5 4 3 2 1

2 2 2 4 4 4 6 6 6 8 8 8

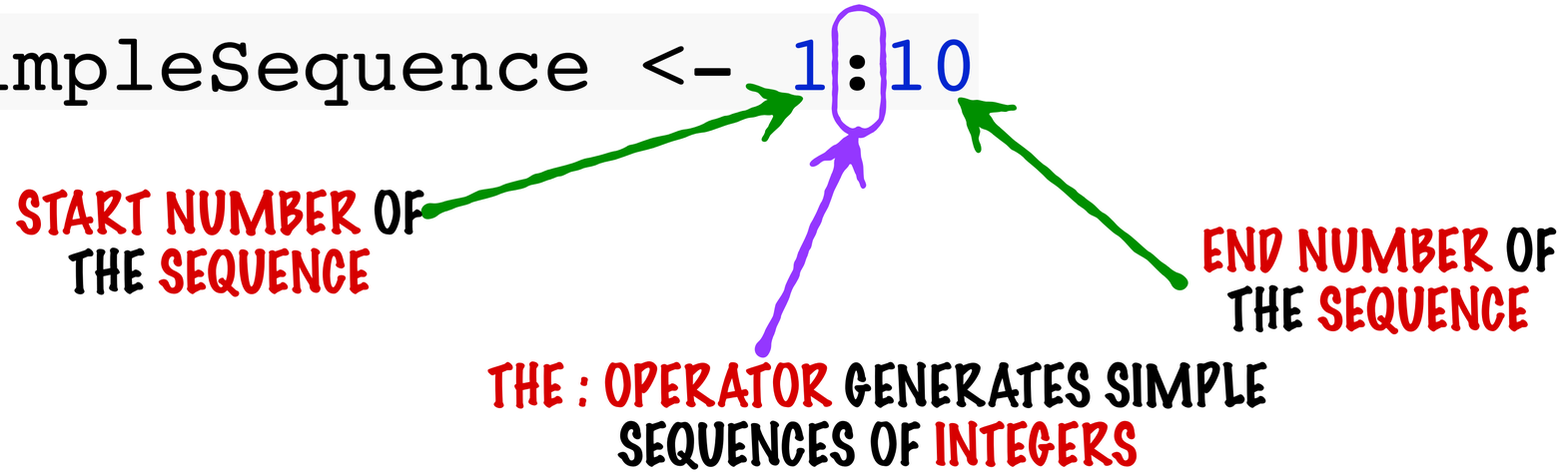
-5.0 -4.8 -4.6 -4.4 -4.2 -4.0 -3.8

```
simpleSequence <- 1:10
simpleSequence
 [1]  1  2  3  4  5  6  7  8  9 10
evenNumberSequence <- 2*1:5
evenNumberSequence
[1]  2  4  6  8 10
reverseSequence <- 10:1
reverseSequence
 [1] 10  9  8  7  6  5  4  3  2  1
repeatSequence <- rep(evenNumberSequence, times = 2, lengt
repeatSequence
 [1]  2  2  2  4  4  4  6  6  6  8  8  8 10 10 10  2  2  2
generalSequence <- seq(from = -5, to = 10, by = 0.2)
generalSequence
  [1] -5.0 -4.8 -4.6 -4.4 -4.2 -4.0 -3.8 -3.6 -3.4 -3.2 -3.
 [15] -2.2 -2.0 -1.8 -1.6 -1.4 -1.2 -1.0 -0.8 -0.6 -0.4 -0.
 [29]  0.6  0.8  1.0  1.2  1.4  1.6  1.8  2.0  2.2  2.4  2.
 [43]  3.4  3.6  3.8  4.0  4.2  4.4  4.6  4.8  5.0  5.2  5.
 [57]  6.2  6.4  6.6  6.8  7.0  7.2  7.4  7.6  7.8  8.0  8.
 [71]  9.0  9.2  9.4  9.6  9.8 10.0
generalSequence2 <- seq(from = -5, to = 10, length.out = 1
generalSequence2
[1] -5.000000 -3.333333 -1.666667  0.000000  1.666667  3.3
[8]  6.666667  8.333333 10.000000
```

EXAMPLE 13: GENERATING SEQUENCES

1 2 3 4 5 6 7 8 9 10

```
simpleSequence <- 1:10
```




```
simpleSequence
```

```
[1]  1  2  3  4  5  6  7  8  9 10
```

EXAMPLE 13: GENERATING SEQUENCES

1 2 3 4 5 6 7 8 9 10

2 4 6 8 10

10 9 8 7 6 5 4 3 2 1

2 2 2 4 4 4 6 6 6 8 8 8

-5.0 -4.8 -4.6 -4.4 -4.2 -4.0 -3.8

```
simpleSequence <- 1:10
simpleSequence
 [1]  1  2  3  4  5  6  7  8  9 10
evenNumberSequence <- 2*1:5
evenNumberSequence
[1]  2  4  6  8 10
reverseSequence <- 10:1
reverseSequence
 [1] 10  9  8  7  6  5  4  3  2  1
repeatSequence <- rep(evenNumberSequence, times = 2, lengt
repeatSequence
 [1]  2  2  2  4  4  4  6  6  6  8  8  8 10 10 10  2  2  2
generalSequence <- seq(from = -5, to = 10, by = 0.2)
generalSequence
  [1] -5.0 -4.8 -4.6 -4.4 -4.2 -4.0 -3.8 -3.6 -3.4 -3.2 -3.
 [15] -2.2 -2.0 -1.8 -1.6 -1.4 -1.2 -1.0 -0.8 -0.6 -0.4 -0.
 [29]  0.6  0.8  1.0  1.2  1.4  1.6  1.8  2.0  2.2  2.4  2.
 [43]  3.4  3.6  3.8  4.0  4.2  4.4  4.6  4.8  5.0  5.2  5.
 [57]  6.2  6.4  6.6  6.8  7.0  7.2  7.4  7.6  7.8  8.0  8.
 [71]  9.0  9.2  9.4  9.6  9.8 10.0
generalSequence2 <- seq(from = -5, to = 10, length.out = 1
generalSequence2
 [1] -5.000000 -3.333333 -1.666667  0.000000  1.666667  3.3
 [8]  6.666667  8.333333 10.000000
```

EXAMPLE 13: GENERATING SEQUENCES

1 2 3 4 5 6 7 8 9 10

2 4 6 8 10

10 9 8 7 6 5 4 3 2 1

2 2 2 4 4 4 6 6 6 8 8 8

-5.0 -4.8 -4.6 -4.4 -4.2 -4.0 -3.8

```
simpleSequence <- 1:10
simpleSequence
 [1]  1  2  3  4  5  6  7  8  9 10
evenNumberSequence <- 2*1:5
evenNumberSequence
 [1]  2  4  6  8 10
reverseSequence <- 10:1
reverseSequence
 [1] 10  9  8  7  6  5  4  3  2  1
repeatSequence <- rep(evenNumberSequence, times = 2, lengt
repeatSequence
 [1]  2  2  2  4  4  4  6  6  6  8  8  8 10 10 10  2  2  2
generalSequence <- seq(from = -5, to = 10, by = 0.2)
generalSequence
  [1] -5.0 -4.8 -4.6 -4.4 -4.2 -4.0 -3.8 -3.6 -3.4 -3.2 -3.
 [15] -2.2 -2.0 -1.8 -1.6 -1.4 -1.2 -1.0 -0.8 -0.6 -0.4 -0.
 [29]  0.6  0.8  1.0  1.2  1.4  1.6  1.8  2.0  2.2  2.4  2.
 [43]  3.4  3.6  3.8  4.0  4.2  4.4  4.6  4.8  5.0  5.2  5.
 [57]  6.2  6.4  6.6  6.8  7.0  7.2  7.4  7.6  7.8  8.0  8.
 [71]  9.0  9.2  9.4  9.6  9.8 10.0
generalSequence2 <- seq(from = -5, to = 10, length.out = 1
generalSequence2
[1] -5.000000 -3.333333 -1.666667  0.000000  1.666667  3.3
[8]  6.666667  8.333333 10.000000
```

EXAMPLE 13: GENERATING SEQUENCES

2 4 6 8 10

```
evenNumberSequence <- 2*1:5
```

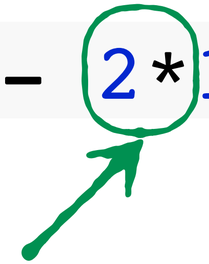
(1, 2, 3, 4, 5)

THE : OPERATOR TAKES PRECEDENCE OVER OTHER OPERATORS

EXAMPLE 13: GENERATING SEQUENCES

2 4 6 8 10

```
evenNumberSequence <- 2*1:5
```



2 * (1, 2, 3, 4, 5)

THE : OPERATOR TAKES PRECEDENCE OVER OTHER OPERATORS

```
evenNumberSequence
```

```
[1]  2  4  6  8 10
```

EXAMPLE 13: GENERATING SEQUENCES

1 2 3 4 5 6 7 8 9 10

2 4 6 8 10

10 9 8 7 6 5 4 3 2 1

2 2 2 4 4 4 6 6 6 8 8 8

-5.0 -4.8 -4.6 -4.4 -4.2 -4.0 -3.8

```
simpleSequence <- 1:10
simpleSequence
 [1]  1  2  3  4  5  6  7  8  9 10
evenNumberSequence <- 2*1:5
evenNumberSequence
 [1]  2  4  6  8 10
reverseSequence <- 10:1
reverseSequence
 [1] 10  9  8  7  6  5  4  3  2  1
repeatSequence <- rep(evenNumberSequence, times = 2, lengt
repeatSequence
 [1]  2  2  2  4  4  4  6  6  6  8  8  8 10 10 10  2  2  2
generalSequence <- seq(from = -5, to = 10, by = 0.2)
generalSequence
  [1] -5.0 -4.8 -4.6 -4.4 -4.2 -4.0 -3.8 -3.6 -3.4 -3.2 -3.
 [15] -2.2 -2.0 -1.8 -1.6 -1.4 -1.2 -1.0 -0.8 -0.6 -0.4 -0.
 [29]  0.6  0.8  1.0  1.2  1.4  1.6  1.8  2.0  2.2  2.4  2.
 [43]  3.4  3.6  3.8  4.0  4.2  4.4  4.6  4.8  5.0  5.2  5.
 [57]  6.2  6.4  6.6  6.8  7.0  7.2  7.4  7.6  7.8  8.0  8.
 [71]  9.0  9.2  9.4  9.6  9.8 10.0
generalSequence2 <- seq(from = -5, to = 10, length.out = 1
generalSequence2
 [1] -5.000000 -3.333333 -1.666667  0.000000  1.666667  3.3
 [8]  6.666667  8.333333 10.000000
```

EXAMPLE 13: GENERATING SEQUENCES

1 2 3 4 5 6 7 8 9 10

2 4 6 8 10

10 9 8 7 6 5 4 3 2 1

2 2 2 4 4 4 6 6 6 8 8 8

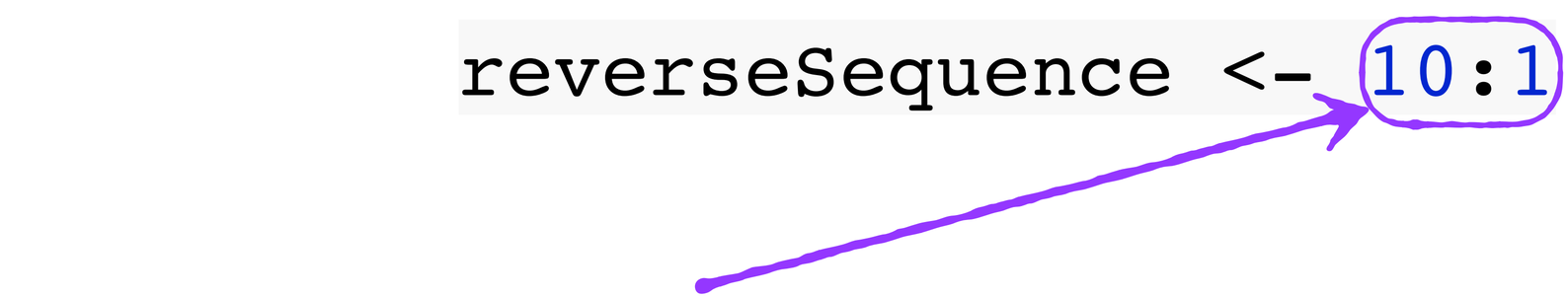
-5.0 -4.8 -4.6 -4.4 -4.2 -4.0 -3.8

```
simpleSequence <- 1:10
simpleSequence
 [1]  1  2  3  4  5  6  7  8  9 10
evenNumberSequence <- 2*1:5
evenNumberSequence
 [1]  2  4  6  8 10
reverseSequence <- 10:1
reverseSequence
 [1] 10  9  8  7  6  5  4  3  2  1
repeatSequence <- rep(evenNumberSequence, times = 2, lengt
repeatSequence
 [1]  2  2  2  4  4  4  6  6  6  8  8  8 10 10 10  2  2  2
generalSequence <- seq(from = -5, to = 10, by = 0.2)
generalSequence
  [1] -5.0 -4.8 -4.6 -4.4 -4.2 -4.0 -3.8 -3.6 -3.4 -3.2 -3.
 [15] -2.2 -2.0 -1.8 -1.6 -1.4 -1.2 -1.0 -0.8 -0.6 -0.4 -0.
 [29]  0.6  0.8  1.0  1.2  1.4  1.6  1.8  2.0  2.2  2.4  2.
 [43]  3.4  3.6  3.8  4.0  4.2  4.4  4.6  4.8  5.0  5.2  5.
 [57]  6.2  6.4  6.6  6.8  7.0  7.2  7.4  7.6  7.8  8.0  8.
 [71]  9.0  9.2  9.4  9.6  9.8 10.0
generalSequence2 <- seq(from = -5, to = 10, length.out = 1
generalSequence2
[1] -5.000000 -3.333333 -1.666667  0.000000  1.666667  3.3
[8]  6.666667  8.333333 10.000000
```

EXAMPLE 13: GENERATING SEQUENCES

10 9 8 7 6 5 4 3 2 1

```
reverseSequence <- 10:1
```



THE : OPERATOR CAN EVEN BE USED TO GENERATE SEQUENCES BACKWARDS

```
reverseSequence
```

```
[1] 10  9  8  7  6  5  4  3  2  1
```

EXAMPLE 13: GENERATING SEQUENCES

1 2 3 4 5 6 7 8 9 10

2 4 6 8 10

10 9 8 7 6 5 4 3 2 1

2 2 2 4 4 4 6 6 6 8 8 8

-5.0 -4.8 -4.6 -4.4 -4.2 -4.0 -3.8

```
simpleSequence <- 1:10
simpleSequence
 [1]  1  2  3  4  5  6  7  8  9 10
evenNumberSequence <- 2*1:5
evenNumberSequence
 [1]  2  4  6  8 10
reverseSequence <- 10:1
reverseSequence
 [1] 10  9  8  7  6  5  4  3  2  1
repeatSequence <- rep(evenNumberSequence, times = 2, lengt
repeatSequence
 [1]  2  2  2  4  4  4  6  6  6  8  8  8 10 10 10  2  2  2
generalSequence <- seq(from = -5, to = 10, by = 0.2)
generalSequence
  [1] -5.0 -4.8 -4.6 -4.4 -4.2 -4.0 -3.8 -3.6 -3.4 -3.2 -3.
 [15] -2.2 -2.0 -1.8 -1.6 -1.4 -1.2 -1.0 -0.8 -0.6 -0.4 -0.
 [29]  0.6  0.8  1.0  1.2  1.4  1.6  1.8  2.0  2.2  2.4  2.
 [43]  3.4  3.6  3.8  4.0  4.2  4.4  4.6  4.8  5.0  5.2  5.
 [57]  6.2  6.4  6.6  6.8  7.0  7.2  7.4  7.6  7.8  8.0  8.
 [71]  9.0  9.2  9.4  9.6  9.8 10.0
generalSequence2 <- seq(from = -5, to = 10, length.out = 1
generalSequence2
[1] -5.000000 -3.333333 -1.666667  0.000000  1.666667  3.3
[8]  6.666667  8.333333 10.000000
```

EXAMPLE 13: GENERATING SEQUENCES

1 2 3 4 5 6 7 8 9 10

2 4 6 8 10

10 9 8 7 6 5 4 3 2 1

2 2 2 4 4 4 6 6 6 8 8 8

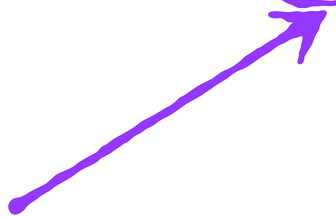
-5.0 -4.8 -4.6 -4.4 -4.2 -4.0 -3.8

```
simpleSequence <- 1:10
simpleSequence
[1]  1  2  3  4  5  6  7  8  9 10
evenNumberSequence <- 2*1:5
evenNumberSequence
[1]  2  4  6  8 10
reverseSequence <- 10:1
reverseSequence
[1] 10  9  8  7  6  5  4  3  2  1
repeatSequence <- rep(evenNumberSequence, times = 2, lengt
repeatSequence
[1]  2  2  2  4  4  4  6  6  6  8  8  8 10 10 10  2  2  2
generalSequence <- seq(from = -5, to = 10, by = 0.2)
generalSequence
 [1] -5.0 -4.8 -4.6 -4.4 -4.2 -4.0 -3.8 -3.6 -3.4 -3.2 -3.
[15] -2.2 -2.0 -1.8 -1.6 -1.4 -1.2 -1.0 -0.8 -0.6 -0.4 -0.
[29]  0.6  0.8  1.0  1.2  1.4  1.6  1.8  2.0  2.2  2.4  2.
[43]  3.4  3.6  3.8  4.0  4.2  4.4  4.6  4.8  5.0  5.2  5.
[57]  6.2  6.4  6.6  6.8  7.0  7.2  7.4  7.6  7.8  8.0  8.
[71]  9.0  9.2  9.4  9.6  9.8 10.0
generalSequence2 <- seq(from = -5, to = 10, length.out = 1
generalSequence2
[1] -5.000000 -3.333333 -1.666667  0.000000  1.666667  3.3
[8]  6.666667  8.333333 10.000000
```

EXAMPLE 13: GENERATING SEQUENCES

2 2 2 4 4 4 6 6 6 8 8 8

```
repeatSequence <- rep(evenNumberSequence, times = 2, length.out = 20, each = 3)
```



THE REP() FUNCTION CAN TAKE A SEQUENCE AND REPLICATE IT IN COMPLEX WAYS

EXAMPLE 13: GENERATING SEQUENCES

THE REP() FUNCTION CAN TAKE A SEQUENCE AND REPLICATE IT IN COMPLEX WAYS

2 2 2 4 4 4 6 6 6 8 8 8

```
repeatSequence <- rep(evenNumberSequence, times = 2, length.out = 20, each = 3)
```

THE SEQUENCE TO BE REPLICATED

2 4 6 8 10

EXAMPLE 13: GENERATING SEQUENCES

THE REP() FUNCTION CAN TAKE A SEQUENCE AND REPLICATE IT IN COMPLEX WAYS

2 2 2 4 4 4 6 6 6 8 8 8

```
repeatSequence <- rep(evenNumberSequence, times = 2, length.out = 20, each = 3)
```

2 4 6 8 10

THE NUMBER OF TIMES EACH ELEMENT IS REPEATED

EXAMPLE 13: GENERATING SEQUENCES

THE REP() FUNCTION CAN TAKE A SEQUENCE AND REPLICATE IT IN COMPLEX WAYS

2 2 2 4 4 4 6 6 6 8 8 8

```
repeatSequence <- rep(evenNumberSequence, times = 2, length.out = 20, each = 3)
```

2 4 6 8 10

2 2 2

THE NUMBER OF TIMES EACH ELEMENT IS REPEATED

EXAMPLE 13: GENERATING SEQUENCES

THE REP() FUNCTION CAN TAKE A SEQUENCE AND REPLICATE IT IN COMPLEX WAYS

2 2 2 4 4 4 6 6 6 8 8 8

```
repeatSequence <- rep(evenNumberSequence, times = 2, length.out = 20, each = 3)
```

2 4 6 8 10

2 2 2 4 4 4

THE NUMBER OF TIMES EACH ELEMENT IS REPEATED

EXAMPLE 13: GENERATING SEQUENCES

THE REP() FUNCTION CAN TAKE A SEQUENCE AND REPLICATE IT IN COMPLEX WAYS

2 2 2 4 4 4 6 6 6 8 8 8

```
repeatSequence <- rep(evenNumberSequence, times = 2, length.out = 20, each = 3)
```

2 4 6 8 10

2 2 2 4 4 4 6 6 6

THE NUMBER OF TIMES EACH ELEMENT IS REPEATED

EXAMPLE 13: GENERATING SEQUENCES

THE REP() FUNCTION CAN TAKE A SEQUENCE AND REPLICATE IT IN COMPLEX WAYS

2 2 2 4 4 4 6 6 6 8 8 8

```
repeatSequence <- rep(evenNumberSequence, times = 2, length.out = 20, each = 3)
```

2 4 6 8 10

2 2 2 4 4 4 6 6 6 8 8 8

THE NUMBER OF TIMES EACH ELEMENT IS REPEATED

EXAMPLE 13: GENERATING SEQUENCES

THE REP() FUNCTION CAN TAKE A SEQUENCE AND REPLICATE IT IN COMPLEX WAYS

2 2 2 4 4 4 6 6 6 8 8 8

```
repeatSequence <- rep(evenNumberSequence, times = 2, length.out = 20, each = 3)
```

2 4 6 8 10

2 2 2 4 4 4 6 6 6 8 8 8 10 10 10

THE NUMBER OF TIMES EACH ELEMENT IS REPEATED

EXAMPLE 13: GENERATING SEQUENCES

THE REP() FUNCTION CAN TAKE A SEQUENCE AND REPLICATE IT IN COMPLEX WAYS

2 2 2 4 4 4 6 6 6 8 8 8

```
repeatSequence <- rep(evenNumberSequence, times = 2, length.out = 20, each = 3)
```

2 4 6 8 10

2 2 2 4 4 4 6 6 6 8 8 8 10 10 10

THEN THE NUMBER OF TIMES THE ENTIRE SEQUENCE IS REPLICATED

EXAMPLE 13: GENERATING SEQUENCES

THE REP() FUNCTION CAN TAKE A SEQUENCE AND REPLICATE IT IN COMPLEX WAYS

2 2 2 4 4 4 6 6 6 8 8 8

```
repeatSequence <- rep(evenNumberSequence, times = 2, length.out = 20, each = 3)
```

2 4 6 8 10

2 2 2 4 4 4 6 6 6 8 8 8 10 10 10

22244466688810 10 10

THEN THE NUMBER OF TIMES THE ENTIRE SEQUENCE IS REPLICATED

EXAMPLE 13: GENERATING SEQUENCES

THE REP() FUNCTION CAN TAKE A SEQUENCE AND REPLICATE IT IN COMPLEX WAYS

2 2 2 4 4 4 6 6 6 8 8 8

```
repeatSequence <- rep(evenNumberSequence, times = 2, length.out = 20, each = 3)
```

2 4 6 8 10

2 2 2 4 4 4 6 6 6 8 8 8 10 10 10

22244466688810101022244466688810101010

THEN THE NUMBER OF TIMES THE ENTIRE SEQUENCE IS REPLICATED

EXAMPLE 13: GENERATING SEQUENCES

THE REP() FUNCTION CAN TAKE A SEQUENCE AND REPLICATE IT IN COMPLEX WAYS

2 2 2 4 4 4 6 6 6 8 8 8

```
repeatSequence <- rep(evenNumberSequence, times = 2, length.out = 20, each = 3)
```

2 4 6 8 10

2 2 2 4 4 4 6 6 6 8 8 10 10 10

THEN TRUNCATE THE SEQUENCE TO THE SPECIFIED LENGTH

22244466688810101022224446668881010

```
repeatSequence
```

```
 [1]  2  2  2  4  4  4  6  6  6  8  8  8 10 10 10  2  2  2  4  4
```

EXAMPLE 13: GENERATING SEQUENCES

THE REP() FUNCTION CAN TAKE A SEQUENCE AND REPLICATE IT IN COMPLEX WAYS

2 2 2 4 4 4 6 6 6 8 8 8

```
repeatSequence <- rep(evenNumberSequence, times = 2, length.out = 20, each = 3)
repeatSequence
 [1]  2  2  2  4  4  4  6  6  6  8  8  8 10 10 10  2  2  2  4  4
```

YOU CAN USE ANY ONE OR MORE OF THESE OPTIONS AT A TIME TO REPLICATE A SEQUENCE

EXAMPLE 13: GENERATING SEQUENCES

1 2 3 4 5 6 7 8 9 10

2 4 6 8 10

10 9 8 7 6 5 4 3 2 1

2 2 2 4 4 4 6 6 6 8 8 8

-5.0 -4.8 -4.6 -4.4 -4.2 -4.0 -3.8

```
simpleSequence <- 1:10
simpleSequence
[1]  1  2  3  4  5  6  7  8  9 10
evenNumberSequence <- 2*1:5
evenNumberSequence
[1]  2  4  6  8 10
reverseSequence <- 10:1
reverseSequence
[1] 10  9  8  7  6  5  4  3  2  1
repeatSequence <- rep(evenNumberSequence, times = 2, lengt
repeatSequence
[1]  2  2  2  4  4  4  6  6  6  8  8  8 10 10 10  2  2  2
generalSequence <- seq(from = -5, to = 10, by = 0.2)
generalSequence
 [1] -5.0 -4.8 -4.6 -4.4 -4.2 -4.0 -3.8 -3.6 -3.4 -3.2 -3.
[15] -2.2 -2.0 -1.8 -1.6 -1.4 -1.2 -1.0 -0.8 -0.6 -0.4 -0.
[29]  0.6  0.8  1.0  1.2  1.4  1.6  1.8  2.0  2.2  2.4  2.
[43]  3.4  3.6  3.8  4.0  4.2  4.4  4.6  4.8  5.0  5.2  5.
[57]  6.2  6.4  6.6  6.8  7.0  7.2  7.4  7.6  7.8  8.0  8.
[71]  9.0  9.2  9.4  9.6  9.8 10.0
generalSequence2 <- seq(from = -5, to = 10, length.out = 1
generalSequence2
[1] -5.000000 -3.333333 -1.666667  0.000000  1.666667  3.3
[8]  6.666667  8.333333 10.000000
```

EXAMPLE 13: GENERATING SEQUENCES

1 2 3 4 5 6 7 8 9 10

2 4 6 8 10

10 9 8 7 6 5 4 3 2 1

2 2 2 4 4 4 6 6 6 8 8 8

-5.0 -4.8 -4.6 -4.4 -4.2 -4.0 -3.8

```
simpleSequence <- 1:10
simpleSequence
 [1]  1  2  3  4  5  6  7  8  9 10
evenNumberSequence <- 2*1:5
evenNumberSequence
 [1]  2  4  6  8 10
reverseSequence <- 10:1
reverseSequence
 [1] 10  9  8  7  6  5  4  3  2  1
repeatSequence <- rep(evenNumberSequence, times = 2, lengt
repeatSequence
 [1]  2  2  2  4  4  4  6  6  6  8  8  8 10 10 10  2  2  2
generalSequence <- seq(from = -5, to = 10, by = 0.2)
generalSequence
  [1] -5.0 -4.8 -4.6 -4.4 -4.2 -4.0 -3.8 -3.6 -3.4 -3.2 -3.
 [15] -2.2 -2.0 -1.8 -1.6 -1.4 -1.2 -1.0 -0.8 -0.6 -0.4 -0.
 [29]  0.6  0.8  1.0  1.2  1.4  1.6  1.8  2.0  2.2  2.4  2.
 [43]  3.4  3.6  3.8  4.0  4.2  4.4  4.6  4.8  5.0  5.2  5.
 [57]  6.2  6.4  6.6  6.8  7.0  7.2  7.4  7.6  7.8  8.0  8.
 [71]  9.0  9.2  9.4  9.6  9.8 10.0
generalSequence2 <- seq(from = -5, to = 10, length.out = 1
generalSequence2
 [1] -5.000000 -3.333333 -1.666667  0.000000  1.666667  3.3
 [8]  6.666667  8.333333 10.000000
```

EXAMPLE 13: GENERATING SEQUENCES

-5.0 -4.8 -4.6 -4.4 -4.2 -4.0 -3.8 .. .. ..

```
generalSequence <- seq(from = -5, to = 10, by = 0.2)

generalSequence2 <- seq(from = -5, to = 10, length.out = 10)

generalSequence3 <- seq(from = -5, to = 10, along.with = evenNumberSequence)
```

THE SEQ() FUNCTION CAN BE USED TO GENERATE SEQUENCES WITH ANY STEP SIZE

EXAMPLE 13: GENERATING SEQUENCES

THE SEQ() FUNCTION CAN BE USED TO GENERATE SEQUENCES WITH ANY STEP SIZE

-5.0 -4.8 -4.6 -4.4 -4.2 -4.0 -3.8 .. .. ..

```
generalSequence <- seq(from = -5, to = 10, by = 0.2)

generalSequence2 <- seq(from = -5, to = 10, length.out = 10)

generalSequence3 <- seq(from = -5, to = 10, along.with = evenNumberSequence)
```

THERE ARE 5 OPTIONS YOU CAN SPECIFY

AND YOU CAN USE ANY ONE OR MORE AT A TIME

EXAMPLE 13: GENERATING SEQUENCES

THE SEQ() FUNCTION CAN BE USED TO GENERATE SEQUENCES WITH ANY STEP SIZE

-5.0 -4.8 -4.6 -4.4 -4.2 -4.0 -3.8 .. .. ..

```
generalSequence <- seq(from = -5, to = 10, by = 0.2)

generalSequence2 <- seq(from = -5, to = 10, length.out = 10)

generalSequence3 <- seq(from = -5, to = 10, along.with = evenNumberSequence)
```

THE START POINT

THERE ARE 5 OPTIONS YOU CAN SPECIFY

AND YOU CAN USE ANY ONE OR MORE AT A TIME

EXAMPLE 13: GENERATING SEQUENCES

THE SEQ() FUNCTION CAN BE USED TO GENERATE SEQUENCES WITH ANY STEP SIZE

-5.0 -4.8 -4.6 -4.4 -4.2 -4.0 -3.8 .. .. ..

```
generalSequence <- seq(from = -5, to = 10, by = 0.2)

generalSequence2 <- seq(from = -5, to = 10, length.out = 10)

generalSequence3 <- seq(from = -5, to = 10, along.with = evenNumberSequence)
```

THERE ARE 5 OPTIONS YOU CAN SPECIFY

AND YOU CAN USE ANY ONE OR MORE AT A TIME

THE START POINT

THE END POINT

EXAMPLE 13: GENERATING SEQUENCES

THE SEQ() FUNCTION CAN BE USED TO GENERATE SEQUENCES WITH ANY STEP SIZE

-5.0 -4.8 -4.6 -4.4 -4.2 -4.0 -3.8 .. .. ..

```
generalSequence <- seq(from = -5, to = 10, by = 0.2)

generalSequence2 <- seq(from = -5, to = 10, length.out = 10)

generalSequence3 <- seq(from = -5, to = 10, along.with = evenNumberSequence)
```

THERE ARE 5 OPTIONS YOU CAN SPECIFY

AND YOU CAN USE ANY ONE OR MORE AT A TIME

THE START POINT

THE END POINT

THE STEP SIZE

EXAMPLE 13: GENERATING SEQUENCES

THE SEQ() FUNCTION CAN BE USED TO GENERATE SEQUENCES WITH ANY STEP SIZE

-5.0 -4.8 -4.6 -4.4 -4.2 -4.0 -3.8 .. .. ..

```
generalSequence <- seq(from = -5, to = 10, by = 0.2)

generalSequence2 <- seq(from = -5, to = 10, length.out = 10)

generalSequence3 <- seq(from = -5, to = 10, along.with = evenNumberSequence)
```

THERE ARE 5 OPTIONS YOU CAN SPECIFY

AND YOU CAN USE ANY ONE OR MORE AT A TIME

THE START POINT

THE END POINT

THE STEP SIZE

EXPLICITLY

THE LENGTH OF THE SEQUENCE

EXAMPLE 13: GENERATING SEQUENCES

THE SEQ() FUNCTION CAN BE USED TO GENERATE SEQUENCES WITH ANY STEP SIZE

-5.0 -4.8 -4.6 -4.4 -4.2 -4.0 -3.8 .. .. ..

```
generalSequence <- seq(from = -5, to = 10, by = 0.2)

generalSequence2 <- seq(from = -5, to = 10, length.out = 10)

generalSequence3 <- seq(from = -5, to = 10, along.with = evenNumberSequence)
```

THERE ARE 5 OPTIONS YOU CAN SPECIFY

AND YOU CAN USE ANY ONE OR MORE AT A TIME

THE START POINT

THE END POINT

THE STEP SIZE

EXPLICITLY

THE LENGTH OF THE SEQUENCE

AS THE LENGTH OF ANOTHER SEQUENCE

EXAMPLE 13: GENERATING SEQUENCES

THE SEQ() FUNCTION CAN BE USED TO GENERATE SEQUENCES WITH ANY STEP SIZE

-5.0 -4.8 -4.6 -4.4 -4.2 -4.0 -3.8 .. .. ..

```
generalSequence <- seq(from = -5, to = 10, by = 0.2)

generalSequence2 <- seq(from = -5, to = 10, length.out = 10)

generalSequence3 <- seq(from = -5, to = 10, along.with = evenNumberSequence)
```

THERE ARE 5 OPTIONS YOU CAN SPECIFY

AND YOU CAN USE ANY ONE OR MORE AT A TIME

USUALLY, YOU MIGHT SPECIFY

THE START POINT
THE END POINT

ONE OR BOTH OF THESE

JUST ONE OF THESE

THE STEP SIZE
THE LENGTH OF THE SEQUENCE
AS THE LENGTH OF ANOTHER SEQUENCE
EXPLICITLY

EXAMPLE 13: GENERATING SEQUENCES

1 2 3 4 5 6 7 8 9 10

2 4 6 8 10

10 9 8 7 6 5 4 3 2 1

2 2 2 4 4 4 6 6 6 8 8 8

-5.0 -4.8 -4.6 -4.4 -4.2 -4.0 -3.8

```
simpleSequence <- 1:10
simpleSequence
 [1]  1  2  3  4  5  6  7  8  9 10
evenNumberSequence <- 2*1:5
evenNumberSequence
 [1]  2  4  6  8 10
reverseSequence <- 10:1
reverseSequence
 [1] 10  9  8  7  6  5  4  3  2  1
repeatSequence <- rep(evenNumberSequence, times = 2, lengt
repeatSequence
 [1]  2  2  2  4  4  4  6  6  6  8  8  8 10 10 10  2  2  2
generalSequence <- seq(from = -5, to = 10, by = 0.2)
generalSequence
  [1] -5.0 -4.8 -4.6 -4.4 -4.2 -4.0 -3.8 -3.6 -3.4 -3.2 -3.
 [15] -2.2 -2.0 -1.8 -1.6 -1.4 -1.2 -1.0 -0.8 -0.6 -0.4 -0.
 [29]  0.6  0.8  1.0  1.2  1.4  1.6  1.8  2.0  2.2  2.4  2.
 [43]  3.4  3.6  3.8  4.0  4.2  4.4  4.6  4.8  5.0  5.2  5.
 [57]  6.2  6.4  6.6  6.8  7.0  7.2  7.4  7.6  7.8  8.0  8.
 [71]  9.0  9.2  9.4  9.6  9.8 10.0
generalSequence2 <- seq(from = -5, to = 10, length.out = 1
generalSequence2
 [1] -5.000000 -3.333333 -1.666667  0.000000  1.666667  3.3
 [8]  6.666667  8.333333 10.000000
```

EXAMPLE 14: CHECKING A CONDITION FOR EACH ELEMENT OF A VECTOR

EXAMPLE 14: CHECKING A CONDITION FOR EACH ELEMENT OF A VECTOR

```
simpleSequence <- 1:4
simpleSequence == 2
```

```
[1] FALSE  TRUE FALSE FALSE
```

EXAMPLE 14: CHECKING A CONDITION FOR EACH ELEMENT OF A VECTOR

```
simpleSequence <- 1:4
```

1 2 3 4

simpleSequence IS A VECTOR

```
simpleSequence == 2
```

```
[1] FALSE  TRUE FALSE FALSE
```

THE ==2 TEST WILL BE APPLIED ON EACH ELEMENT OF THE VECTOR

THE RESULT IS A LOGICAL VECTOR OF THE SAME LENGTH AS simpleSequence

I.E. EACH ELEMENT OF THE RESULT IS A LOGICAL (EITHER TRUE OR FALSE)

EXAMPLE 15: FINDING THE LENGTHS OF EACH STRING IN A VECTOR OF STRINGS

EXAMPLE 15: FINDING THE LENGTHS OF EACH STRING IN A VECTOR OF STRINGS

```
stringSequence <- c("A", "B", "C", "D", "E", "F", "G", "H")
nchar(stringSequence)
```

```
[1] 1 1 1 1 1 1 1 1
```

EXAMPLE 15: FINDING THE LENGTHS OF EACH STRING IN A VECTOR OF STRINGS

```
stringSequence <- c("A", "B", "C", "D", "E", "F", "G", "H")
nchar(stringSequence)
```

NCHAR() IS A FUNCTION THAT WILL TELL YOU THE LENGTH OF A STRING

EXAMPLE 15: FINDING THE LENGTHS OF EACH STRING IN A VECTOR OF STRINGS

```
stringSequence <- c("A", "B", "C", "D", "E", "F", "G", "H")
nchar(stringSequence)
```

NCHAR() IS A FUNCTION THAT WILL TELL YOU THE LENGTH OF A STRING

stringSequence IS A CHARACTER VECTOR I.E. EACH ELEMENT IS A STRING

THE FUNCTION IS APPLIED TO EACH ELEMENT OF THE VECTOR

EXAMPLE 15: FINDING THE LENGTHS OF EACH STRING IN A VECTOR OF STRINGS

```
stringSequence <- c("A", "B", "C", "D", "E", "F", "G", "H")
nchar(stringSequence)
```

```
[1] 1 1 1 1 1 1 1 1
```

NCHAR() IS A FUNCTION THAT WILL TELL YOU THE LENGTH OF A STRING

stringSequence IS A CHARACTER VECTOR I.E. EACH ELEMENT IS A STRING

THE FUNCTION IS APPLIED TO EACH ELEMENT OF THE VECTOR

EXAMPLE 16: GENERATE THE SEQUENCE

A1, B2, C3, D4, E1, F2, G3, H4

EXAMPLE 16: GENERATE THE SEQUENCE A1, B2, C3, ...

```
simpleSequence <- 1:4
stringSequence <- c("A", "B", "C", "D", "E", "F", "G", "H")
funkySequence <- paste(stringSequence, simpleSequence, sep="")
```

paste() WILL COMBINE ANY NUMBER OF VARIABLES INTO A STRING

EXAMPLE 9: MORE FUN WITH RECYCLING

```
funkySequence <- paste(stringSequence, simpleSequence, sep="")
```

paste() WILL COMBINE
ANY NUMBER OF
VARIABLES INTO A STRING

SEPARATED BY THE
SPECIFIED DELIMITER

EXAMPLE 16: GENERATE THE SEQUENCE A1, B2, C3, ...

```
funkySequence <- paste(stringSequence, simpleSequence, sep=" "
```

paste() WILL COMBINE ANY NUMBER OF VARIABLES INTO A STRING

SEPARATED BY THE SPECIFIED DELIMITER

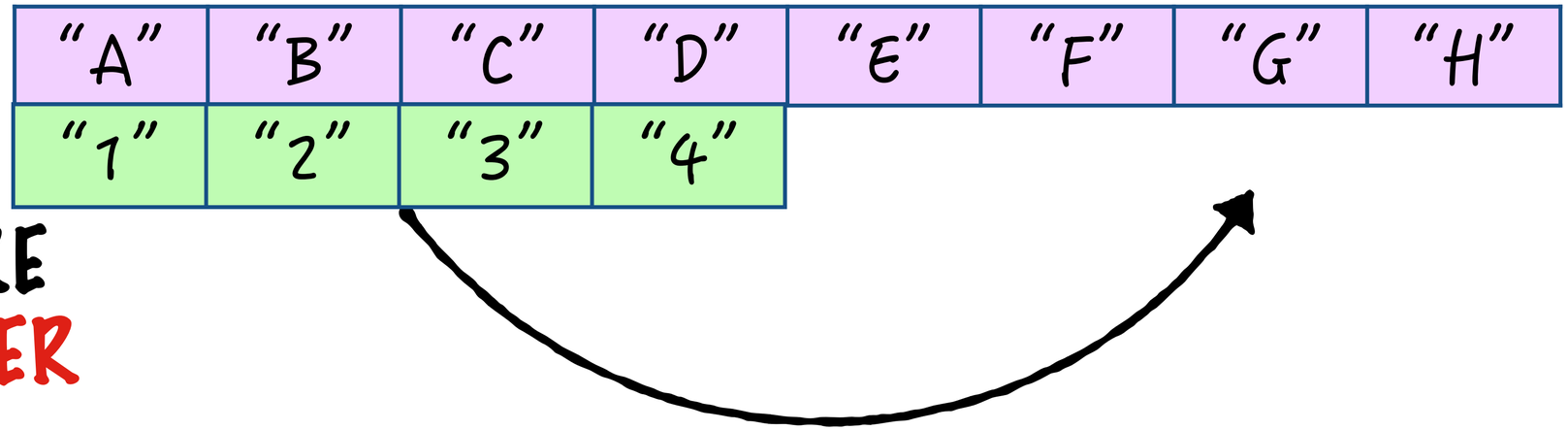
EXAMPLE 16: GENERATE THE SEQUENCE A1, B2, C3, ...

```
funkySequence <- paste(stringSequence, simpleSequence, sep=" "
```

"A"	"B"	"C"	"D"	"E"	"F"	"G"	"H"

EXAMPLE 16: GENERATE THE SEQUENCE A1, B2, C3, ...

```
funkySequence <- paste(stringSequence, simpleSequence, sep=" "
```



FIRST THE NUMBERS ARE CONVERTED TO CHARACTER

THEN THE SHORTER VECTOR IS RECYCLED TWICE

```
funkySequence
```

```
[1] "A1" "B2" "C3" "D4" "E1" "F2" "G3" "H4"
```

EXAMPLE 17: VECTOR INDEXING BASED ON POSITION

[]

ARE USED TO SELECT ELEMENTS OF A VECTOR

[] ARE USED TO SELECT ELEMENTS OF A VECTOR

```
stringSequence[6]
```

WILL GIVE YOU THE 6TH ELEMENT OF stringSequence

IMPORTANT!!! :

INDEXING IN R STARTS FROM 1

WITHIN THE [] OPERATOR
THERE CAN BE ANOTHER VECTOR

THIS VECTOR CAN BE MADE UP OF

POSITIVE INTEGERS

TO SPECIFY THE POSITION OF THE ELEMENTS TO BE SELECTED

WITHIN THE [] OPERATOR THERE CAN BE ANOTHER VECTOR

THIS VECTOR CAN BE MADE UP OF

POSITIVE INTEGERS — TO SPECIFY THE POSITION OF THE ELEMENTS TO BE SELECTED

NEGATIVE INTEGERS

TO SPECIFY THE POSITION OF THE ELEMENTS THAT SHOULD NOT BE SELECTED

WITHIN THE [] OPERATOR THERE CAN BE ANOTHER VECTOR

THIS VECTOR CAN BE MADE UP OF

POSITIVE INTEGERS — TO SPECIFY THE POSITION OF THE ELEMENTS TO BE SELECTED

NEGATIVE INTEGERS — TO SPECIFY THE POSITION OF THE ELEMENTS THAT SHOULD NOT BE SELECTED

LET'S CREATE A VECTOR

```
mySequence <- 3*1:5
mySequence
[1]  3  6  9 12 15
```

LET'S CREATE A VECTOR

```
mySequence <- 3*1:5
mySequence
[1]  3  6  9 12 15
```

SELECT THE FIRST ELEMENT

```
mySequence[1]
[1] 3
```

LET'S CREATE A VECTOR

```
mySequence <- 3*1:5
mySequence
[1]  3  6  9 12 15
```

HOW WILL YOU SELECT

3 6 9 12 15

```
mySequence[2:4]
[1]  6  9 12
```

LET'S CREATE A VECTOR

```
mySequence <- 3*1:5
mySequence
[1]  3  6  9 12 15
```

1 3

WHAT DOES THIS DO?

1, 3, 1, 3, 1, 3, 1, 3, 1, 3

```
mySequence[rep(c(1,3), times = 5)]
 [1]  3  9  3  9  3  9  3  9  3  9
```

REPLICATE (1,3) 5 TIMES

LET'S CREATE A VECTOR

```
mySequence <- 3*1:5
mySequence
[1]  3  6  9 12 15
```

3 4

```
mySequence[c(-3, -4)]
```

DO NOT SELECT THE 3RD AND 4TH ELEMENTS

```
[1]  3  6 15
```

EXAMPLE 18: VECTOR INDEXING BASED ON CONDITIONS

LET'S CREATE A VECTOR

```
mySequence <- 3*1:5
mySequence
 [1]  3  6  9 12 15
```

F, F, F, T, T

```
mySequence[mySequence > 10]
```

A LOGICAL VECTOR OF THE SAME LENGTH AS mySequence

LET'S SELECT THE ELEMENTS OF THE VECTOR THAT SATISFY THE CONDITION >10

LET'S CREATE A VECTOR

```
mySequence <- 3*1:5
mySequence
 [1]  3  6  9 12 15
```

F, F, F, T, T

```
mySequence[mySequence > 10]
```

```
[1] 12 15
```

DROP ELEMENTS CORRESPONDING TO FALSE, AND INCLUDE THOSE CORRESPONDING TO TRUE

LET'S CREATE A VECTOR

```
mySequence <- 3*1:5
mySequence
 [1]  3  6  9 12 15
```

THIS CONDITION WILL SELECT THE ELEMENTS WHICH ARE NOT EQUAL TO 9

```
mySequence[mySequence != 9] <- -mySequence[mySequence !=9]
```

LET'S INVERT THE SIGN OF THE ELEMENTS WHICH ARE NOT EQUAL TO 9

LET'S CREATE A VECTOR

```
mySequence <- 3*1:5
mySequence
 [1]  3  6  9 12 15
```

-3, -6, -12, -15

```
mySequence[mySequence != 9] <- -mySequence[mySequence !=9]
```

ASSIGN THIS VECTOR TO THE ELEMENTS SPECIFIED BY THE LHS

LET'S CREATE A VECTOR

```
mySequence <- 3*1:5
mySequence
[1]  3  6  9 12 15
```

-3, -6, -12, -15

```
mySequence[mySequence != 9] <- -mySequence[mySequence !=9]
```

ASSIGN THIS VECTOR TO THE ELEMENTS SPECIFIED BY THE LHS

LET'S CREATE A VECTOR

```
mySequence <- 3*1:5
mySequence
 [1]  3  6  9 12 15
```

-3, -6, -9, -15

```
mySequence[mySequence != 9] <- -mySequence[mySequence !=9]
mySequence
[1] -3  -6   9 -12 -15
```

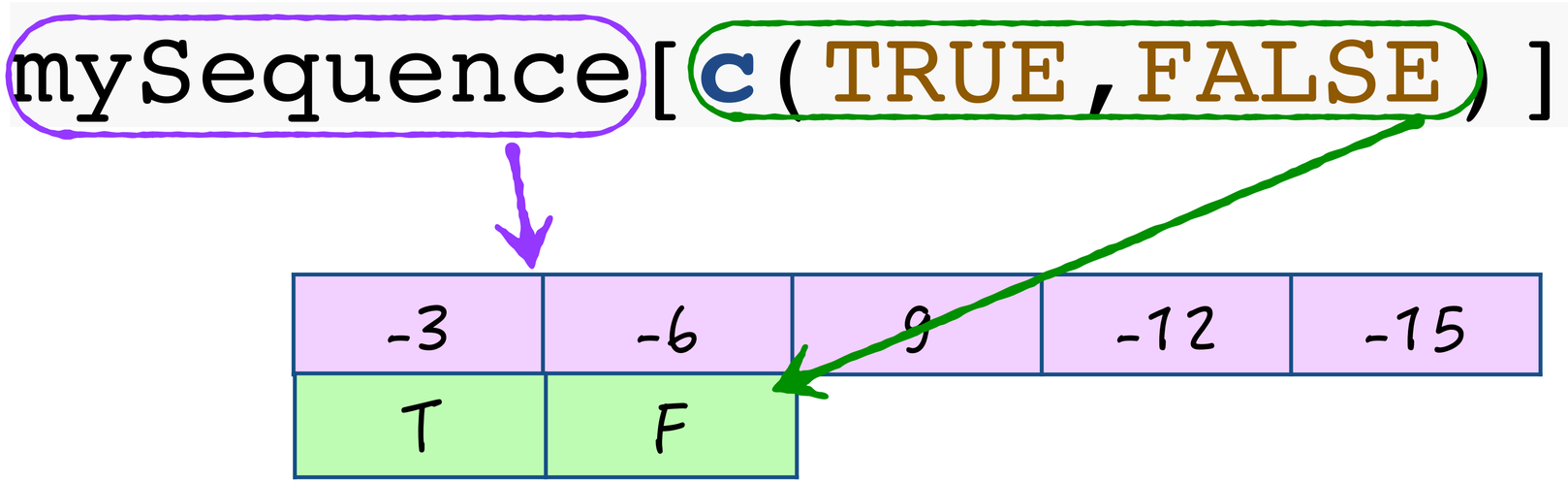
```
mySequence
```

```
[1] -3  -6   9 -12 -15
```

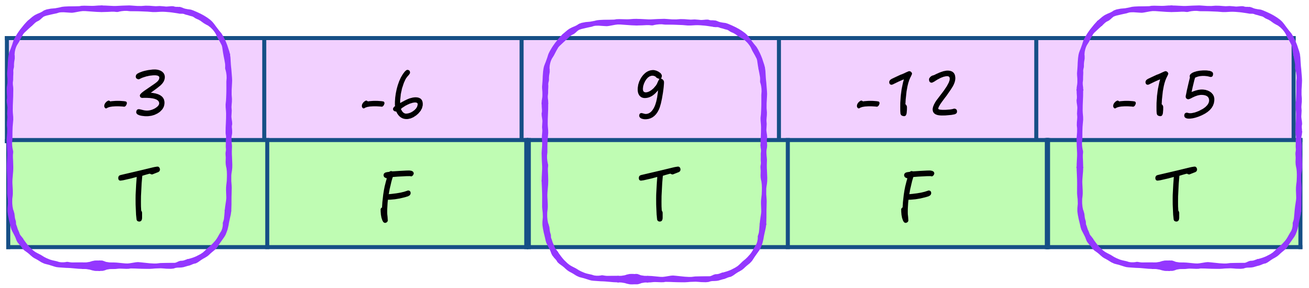
```
mySequence[c(TRUE,FALSE)]
```

WHEN YOU USE A LOGICAL VECTOR FOR INDEXING, THE INDEX VECTOR HAS TO BE OF THE SAME LENGTH AS THE ORIGINAL VECTOR

IF NOT, IT IS RECYCLED

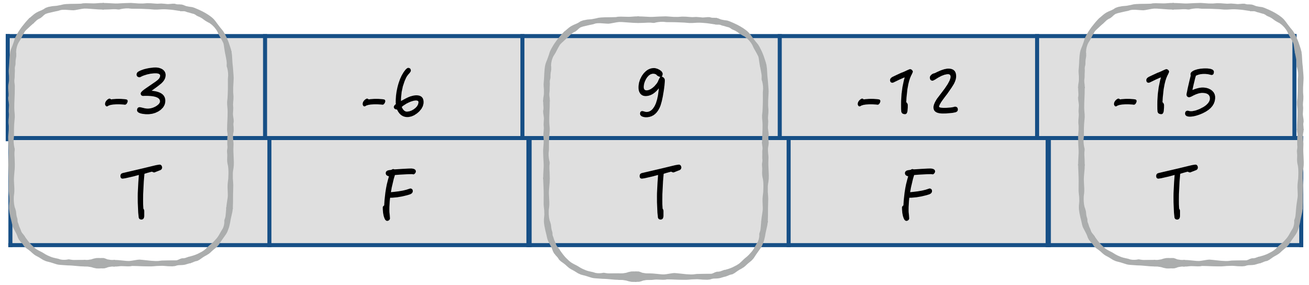

mySequence[c(TRUE,FALSE)]
-3
-6
9
-12
-15
T
F

```
mySequence[c(TRUE,FALSE)]
```



```
mySequence[c(TRUE,FALSE)]
```

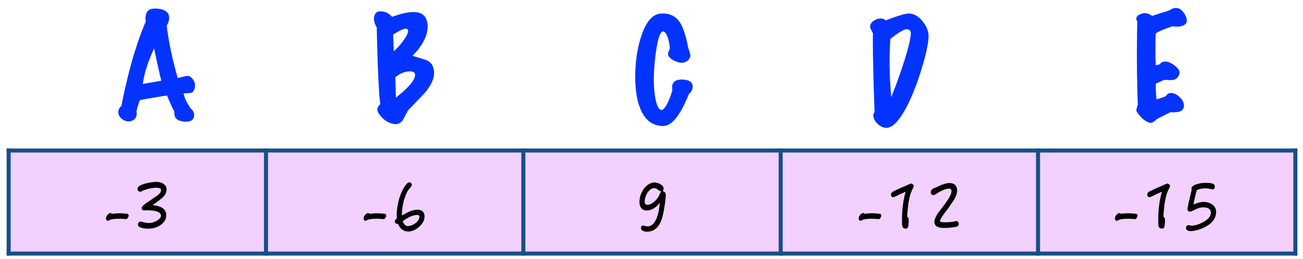
```
[1] -3   9  -15
```



EXAMPLE 19: ASSIGNING NAMES TO VECTOR ELEMENTS

EACH ELEMENT IN A VECTOR CAN BE GIVEN A "NAME"

```
names(mySequence) <- c("A","B","C","D","E")
```



```
names(mySequence) <- c("A","B","C","D","E")
```

```
mySequence[c("A","C")]
```

```
A  C
-3  9
```

THEN ELEMENTS CAN BE INDEXED BY THEIR NAMES