# Lecture 2-3

# Flow Control; Lists

Week 2 Friday

Miles Chen, PhD

Adapted from *Think Python* by Allen B. Downey and *A Whirlwind Tour of Python* by Jake VanderPlas

# if-elif-else

The basic statement for control the flow of execution are the if-elif-else conditional statements.

- There is no need to use parenthesis in the conditional statements.
- Use a colon to end the conditional statement.
- Lines indented after the colon are associated with the if statement.
- When there is no longer indentation, the lines are no longer associated with the if statement.
- `elif` (else if) and `else` must be on the same level of indentation as the first `if` statement.

In [1]:

```python
x = -3

if x == 0:
    print(x, 'is zero')
elif x > 0:
    print(x, 'is positive')
else:
    print(x, 'must be negative')
```

```
-3 must be negative
```

Like other languages, the `elif` or `else` statements are only executed if the original `if` statement is false

In [2]:

```
x = 100

if x > 0:
    print(x, 'is positive')
elif x > 3:
    print(x, 'is greater than 3')  # will not get executed
else:
    print(x, 'is zero or negative')
```

```
100 is positive
```

# Nested Conditionals

You can nest conditionals, but they can be hard to read and should be avoided when possible.

# Nested Conditionals

You can nest conditionals, but they can be hard to read and should be avoided when possible.

In [3]:

```
x = 5
if 0 < x:
    if x < 10:
        print('x is a positive single-digit number.')
```

```
x is a positive single-digit number.
```

# Nested Conditionals

You can nest conditionals, but they can be hard to read and should be avoided when possible.

In [3]:
```
x = 5
if 0 < x:
    if x < 10:
        print('x is a positive single-digit number.')
```

```
x is a positive single-digit number.
```

In [4]:
```
# better alternative
if 0 < x and x < 10:
    print('x is a positive single-digit number.')
```

```
x is a positive single-digit number.
```

## Nested Conditionals

You can nest conditionals, but they can be hard to read and should be avoided when possible.

In [3]:
```
x = 5
if 0 < x:
    if x < 10:
        print('x is a positive single-digit number.')
```

```
x is a positive single-digit number.
```

In [4]:
```
# better alternative
if 0 < x and x < 10:
    print('x is a positive single-digit number.')
```

```
x is a positive single-digit number.
```

In [5]:
```
# concise format:
if 0 < x < 10:
    print('x is a positive single-digit number.')
```

```
x is a positive single-digit number.
```

# Recursion

When you write a recursive function, the function calls itself inside the function.

When you write a recursive function, there should always be a base case that does not call the function recursively. This will end the function to avoid it from running forever.

# Recursion

When you write a recursive function, the function calls itself inside the function.

When you write a recursive function, there should always be a base case that does not call the function recursively. This will end the function to avoid it from running forever.

In [6]:

```
def countdown(n):
    if n <= 0:
        print('Blastoff!')
    else:
        print(n)
        countdown(n - 1)
```

# Recursion

When you write a recursive function, the function calls itself inside the function.

When you write a recursive function, there should always be a base case that does not call the function recursively. This will end the function to avoid it from running forever.

In [6]:
```
def countdown(n):
    if n <= 0:
        print('Blastoff!')
    else:
        print(n)
        countdown(n - 1)
```

In [7]:
```
countdown(3)
```

```
3
2
1
Blastoff!
```

- The execution of countdown begins with n=3, and since n is greater than 0, it prints the value 3, and then calls itself with n=2
    - The execution of countdown begins with n=2, and since n is greater than 0, it prints the value 2, and then calls itself with n = 1
        - The execution of countdown begins with n=1, and since n is greater than 0, it prints the value 1, and then calls itself with n = 0
            - The execution of countdown begins with n=0, and since n is not greater than 0, it prints the word, “Blastoff!” and then returns.
        - The countdown that got n=1 returns.
    - The countdown that got n=2 returns.
- The countdown that got n=3 returns.

In [8]:

```
# another example
# a function that prints a string n times
```

In [8]:

```
# another example
# a function that prints a string n times
```

In [9]:

```
def print_n(s, n):
    if n <= 0:
        return None # exits the function
    print(s)
    print_n(s, n - 1)
```

In [10]:

```
print_n("hello", 3)
```

```
hello
hello
hello
```

Factorial function is also a good candidate for recursion.

- 4! = 4 * 3!
- 3! = 3 * 2!
- 2! = 2 * 1!
- 1! = 1 * 0!
- 0! = 1

Factorial function is also a good candidate for recursion.

- 4! = 4 * 3!
- 3! = 3 * 2!
- 2! = 2 * 1!
- 1! = 1 * 0!
- 0! = 1

In [11]:

```
def factorial(n):
    if n <= 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Factorial function is also a good candidate for recursion.

- 4! = 4 * 3!
- 3! = 3 * 2!
- 2! = 2 * 1!
- 1! = 1 * 0!
- 0! = 1

In [11]:

```
def factorial(n):
    if n <= 0:
        return 1
    else:
        return n * factorial(n - 1)
```

In [12]:

```
factorial(4)
```

Out[12]: 24

Factorial function is also a good candidate for recursion.

- 4! = 4 * 3!
- 3! = 3 * 2!
- 2! = 2 * 1!
- 1! = 1 * 0!
- 0! = 1

In [11]:
```
def factorial(n):
    if n <= 0:
        return 1
    else:
        return n * factorial(n - 1)
```

In [12]:
```
factorial(4)
```

Out[12]:
```
24
```

In [13]:
```
factorial(5)
```

Out[13]:
```
120
```

# for loops

It is technically possible to accomplish all forms of repetition with recursion only. However, repetition can also be achieved using other coding forms like loops.

The most basic loop type is the for loop, which will repeat the associated lines of code for each value in an iterable. Python has several iterable data structures (list, tuple, range, strings, etc.) which will be covered in more detail later.

# for loops

It is technically possible to accomplish all forms of repetition with recursion only. However, repetition can also be achieved using other coding forms like loops.

The most basic loop type is the for loop, which will repeat the associated lines of code for each value in an iterable. Python has several iterable data structures (list, tuple, range, strings, etc.) which will be covered in more detail later.

In [14]:

```
values = [5, 7, 2, 1]
y = 0
for x in values:
    print(x)
    y += x  # short for y = y + x
    print('running sum is:', y)
```

```
5
running sum is: 5
7
running sum is: 12
2
running sum is: 14
1
running sum is: 15
```

In [15]:

```
for a in "ucla ucla":
    print(a.upper() + "!")
```

```
U!
C!
L!
A!
 !
U!
C!
L!
A!
```

# The range object

If you want just a sequence of numbers, you can use a `range()` object.

`range(10)` is similar to writing 0:9 in R. It creates a range of indexes that is 10 items long and begins with index 0.

The general format is

```
range( start , end , step size)
```

by default, the range will begin at the start value, increment by step size, and go up to but not include the end value. If you only specify one integer value, it will assume start = 0 and step size is 1.

# The range object

If you want just a sequence of numbers, you can use a `range()` object.

`range(10)` is similar to writing 0:9 in R. It creates a range of indexes that is 10 items long and begins with index 0.

The general format is

`range( start , end , step size)`

by default, the range will begin at the start value, increment by step size, and go up to but not include the end value. If you only specify one integer value, it will assume start = 0 and step size is 1.

In [16]:
```
range(10)
```

Out[16]:
```
range(0, 10)
```

# The range object

If you want just a sequence of numbers, you can use a `range()` object.

`range(10)` is similar to writing 0:9 in R. It creates a range of indexes that is 10 items long and begins with index 0.

The general format is

`range( start , end , step size)`

by default, the range will begin at the start value, increment by step size, and go up to but not include the end value. If you only specify one integer value, it will assume start = 0 and step size is 1.

In [16]:

```
range(10)
```

Out[16]:

```
range(0, 10)
```

In [17]:

```
for i in range(10):
    print(i, end = ' ')
    # the end argument tells python to use a space rather than a new line
```

```
0 1 2 3 4 5 6 7 8 9
```

In [18]:

```
range(5,10)  # creates a range from 5 up to but not including 10
```

Out[18]:

```
range(5, 10)
```

In [18]:
```
range(5,10)  # creates a range from 5 up to but not including 10
```

Out[18]:
```
range(5, 10)
```

In [19]:
```
list(range(5,10))  # if you want to see the actual values, throw in list
```

Out[19]:
```
[5, 6, 7, 8, 9]
```

In [18]:
```
range(5,10)  # creates a range from 5 up to but not including 10
```

Out[18]:
```
range(5, 10)
```

In [19]:
```
list(range(5,10))  # if you want to see the actual values, throw in list
```

Out[19]:
```
[5, 6, 7, 8, 9]
```

In [20]:
```
list(range(0, 20, 2))  # range from 0 to 20 by 2
```

Out[20]:
```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

In [18]:
```
range(5,10)  # creates a range from 5 up to but not including 10
```

Out[18]:
```
range(5, 10)
```

In [19]:
```
list(range(5,10))  # if you want to see the actual values, throw in list
```

Out[19]:
```
[5, 6, 7, 8, 9]
```

In [20]:
```
list(range(0, 20, 2))  # range from 0 to 20 by 2
```

Out[20]:
```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

In [21]:
```
list(range(0, 21, 2))
```

Out[21]:
```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

In [18]:
```
range(5,10)  # creates a range from 5 up to but not including 10
```
Out[18]:
```
range(5, 10)
```
In [19]:
```
list(range(5,10))  # if you want to see the actual values, throw in list
```
Out[19]:
```
[5, 6, 7, 8, 9]
```
In [20]:
```
list(range(0, 20, 2))  # range from 0 to 20 by 2
```
Out[20]:
```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```
In [21]:
```
list(range(0, 21, 2))
```
Out[21]:
```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```
In [22]:
```
list(range(0, 20.1, 2))  #does not accept floats as arguments
```
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-22-7c865feae284> in <module>
----> 1 list(range(0, 20.1, 2))  #does not accept floats as arguments

TypeError: 'float' object cannot be interpreted as an integer
```

In [23]:

```
list(range(10,5,-1)) # need to specify a negative step
```

Out[23]:

```
[10, 9, 8, 7, 6]
```

In [23]:
```
list(range(10,5,-1)) # need to specify a negative step
```
Out[23]:
```
[10, 9, 8, 7, 6]
```
In [24]:
```
list(range(10,5)) # otherwise you get no values in your list
```
Out[24]:
```
[]
```

# while loops

Another common loop is the while loop. It repeats the associated code until the conditional statement is False

# while loops

Another common loop is the while loop. It repeats the associated code until the conditional statement is False

In [25]:

```
i = 0
while i < 10:
    print(i, end=' ')
    i += 1
```

```
0 1 2 3 4 5 6 7 8 9
```

# while loops

Another common loop is the while loop. It repeats the associated code until the conditional statement is False

In [25]:

```
i = 0
while i < 10:
    print(i, end=' ')
    i += 1
```

```
0 1 2 3 4 5 6 7 8 9
```

In [26]:

```
i
```

Out[26]:

```
10
```

# break and continue

- The break statement breaks-out of the loop entirely
- The continue statement skips the remainder of the current loop, and goes to the next iteration

# break and continue

- The break statement breaks-out of the loop entirely
- The continue statement skips the remainder of the current loop, and goes to the next iteration

In [27]:

```
for n in range(20):
    # if the remainder of n / 2 is 0, skip the rest of the loop
    if n % 2 == 0:
        continue
    print(n, end=' ')
```

```
1 3 5 7 9 11 13 15 17 19
```

an example to create fibonacci numbers

## break and continue

- The break statement breaks-out of the loop entirely
- The continue statement skips the remainder of the current loop, and goes to the next iteration

In [27]:

```
for n in range(20):
    # if the remainder of n / 2 is 0, skip the rest of the loop
    if n % 2 == 0:
        continue
    print(n, end=' ')
```

```
1 3 5 7 9 11 13 15 17 19
```

an example to create fibonacci numbers

In [28]:

```
a, b = 0, 1   # you can assign multiple values using tuples
amax = 100    # set a maximum value
L = []

while True:    # the while True will run forever until it reaches a break
    (a, b) = (b, a + b)
    if a > amax:
        break
    L.append(a)

print(L)
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

# Boolean expressions

All conditional statements rely on the use of boolean expressions.

# Boolean expressions

All conditional statements rely on the use of boolean expressions.

In [29]:
```
5 == 5
```

Out[29]:
```
True
```

# Boolean expressions

All conditional statements rely on the use of boolean expressions.

In [29]:
```
5 == 5
```

Out[29]:
```
True
```

In [30]:
```
5 == 6
```

Out[30]:
```
False
```

# Boolean expressions

All conditional statements rely on the use of boolean expressions.

In [29]:
```
5 == 5
```

Out[29]:
```
True
```

In [30]:
```
5 == 6
```

Out[30]:
```
False
```

In [31]:
```
5 != 6
```

Out[31]:
```
True
```

In [32]:
```
5 > 6
```

Out[32]:
```
False
```

In [32]:
```
5 > 6
```
Out[32]:
```
False
```
In [33]:
```
5 < 6
```
Out[33]:
```
True
```

In [32]:
```
5 > 6
```
Out[32]:
```
False
```
In [33]:
```
5 < 6
```
Out[33]:
```
True
```
In [34]:
```
5 >= 6
```
Out[34]:
```
False
```

In [32]:
```
5 > 6
```
Out[32]: False

In [33]:
```
5 < 6
```
Out[33]: True

In [34]:
```
5 >= 6
```
Out[34]: False

In [35]:
```
5 <= 6
```
Out[35]: True

In [36]:
```
"5" == 5
```

Out[36]:
```
False
```

When comparing strings, greater than or less than is determined by alphabetical order, with things coming earlier in the alphabet being "less than" things later in the alphabet.

All upper case letters come before all lower case letters

In [36]:
```
"5" == 5
```

Out[36]:
```
False
```

When comparing strings, greater than or less than is determined by alphabetical order, with things coming earlier in the alphabet being "less than" things later in the alphabet.

All upper case letters come before all lower case letters

In [37]:
```
"A" < "B"
```

Out[37]:
```
True
```

In [36]:
```
"5" == 5
```

Out[36]: False

When comparing strings, greater than or less than is determined by alphabetical order, with things coming earlier in the alphabet being "less than" things later in the alphabet.

All upper case letters come before all lower case letters

In [37]:
```
"A" < "B"
```

Out[37]: True

In [38]:
```
"A" <"a"
```

Out[38]: True

In [36]:
```
"5" == 5
```

Out[36]:
```
False
```

When comparing strings, greater than or less than is determined by alphabetical order, with things coming earlier in the alphabet being "less than" things later in the alphabet.

All upper case letters come before all lower case letters

In [37]:
```
"A" < "B"
```

Out[37]:
```
True
```

In [38]:
```
"A" <"a"
```

Out[38]:
```
True
```

In [39]:
```
"Z" < "a"
```

Out[39]:
```
True
```

In [36]:
```python
"5" == 5
```

Out[36]:
```
False
```

When comparing strings, greater than or less than is determined by alphabetical order, with things coming earlier in the alphabet being "less than" things later in the alphabet.

All upper case letters come before all lower case letters

In [37]:
```python
"A" < "B"
```

Out[37]:
```
True
```

In [38]:
```python
"A" <"a"
```

Out[38]:
```
True
```

In [39]:
```python
"Z" < "a"
```

Out[39]:
```
True
```

In [40]:
```python
"a" < "z"
```

Out[40]:
```
True
```

# Logical operators

`and` `or` `not` are written in lowercase

# Logical operators

`and` `or` `not` are written in lowercase

In [41]:
```
True and True
```

Out[41]:
```
True
```

# Logical operators

`and` `or` `not` are written in lowercase

In [41]:
```
True and True
```

Out[41]:
```
True
```

In [42]:
```
True and False
```

Out[42]:
```
False
```

# Logical operators

`and` `or` `not` are written in lowercase

In [41]:
```
True and True
```
Out[41]:
```
True
```
In [42]:
```
True and False
```
Out[42]:
```
False
```
In [43]:
```
True or False
```
Out[43]:
```
True
```

# Logical operators

`and` `or` `not` are written in lowercase

In [41]:
```python
True and True
```
Out[41]:
```
True
```
In [42]:
```python
True and False
```
Out[42]:
```
False
```
In [43]:
```python
True or False
```
Out[43]:
```
True
```
In [44]:
```python
not True
```
Out[44]:
```
False
```

# Logical operators

`and` `or` `not` are written in lowercase

In [41]:
```
True and True
```
Out[41]:
```
True
```

In [42]:
```
True and False
```
Out[42]:
```
False
```

In [43]:
```
True or False
```
Out[43]:
```
True
```

In [44]:
```
not True
```
Out[44]:
```
False
```

In [45]:
```
not False
```
Out[45]:
```
True
```

In [46]:

```
False or not False
```

Out[46]:

```
True
```

In [46]:
```
False or not False
```

Out[46]:
```
True
```

In [47]:
```
True and not False
```

Out[47]:
```
True
```

The idiom `x % y == 0` is a way to check if x is divisible by y.

In [46]:
```python
False or not False
```

Out[46]:
```
True
```

In [47]:
```python
True and not False
```

Out[47]:
```
True
```

The idiom `x % y == 0` is a way to check if x is divisible by y.

In [48]:
```python
n = 6
n % 2 == 0 and n % 3 == 0
```

Out[48]:
```
True
```

In [46]:
```
False or not False
```
Out[46]:
```
True
```
In [47]:
```
True and not False
```
Out[47]:
```
True
```
The idiom `x % y == 0` is a way to check if x is divisible by y.

In [48]:
```
n = 6
n % 2 == 0 and n % 3 == 0
```
Out[48]:
```
True
```
In [49]:
```
n = 8
n % 2 == 0 and n % 3 == 0
```
Out[49]:
```
False
```

In [46]:
```python
False or not False
```
Out[46]:
```
True
```
In [47]:
```python
True and not False
```
Out[47]:
```
True
```
The idiom `x % y == 0` is a way to check if x is divisible by y.

In [48]:
```python
n = 6
n % 2 == 0 and n % 3 == 0
```
Out[48]:
```
True
```
In [49]:
```python
n = 8
n % 2 == 0 and n % 3 == 0
```
Out[49]:
```
False
```
In [50]:
```python
n = 8
n % 2 == 0 or n % 3 == 0
```
Out[50]:
```
True
```

# A little bit on strategies for writing functions

When writing a function, I advise against going straight to writing the function.

You should first write code in the global environment to achieve the desired task.

Once you achieve this, then you can encapsulate the lines within a function.

```
# pseudo code for drawing a square

go_forward(100) # value in px
turn_left(90) # value in degrees
go_forward(100)
turn_left(90)
go_forward(100)
turn_left(90)
go_forward(100)
turn_left(90)
```

# Encapsulation

At the most basic level, a function encapsulates a few lines of code. This associates a name with statements and allows us to reuse the code.

For example let's say we wanted to write some functions for drawing shapes:

# Encapsulation

At the most basic level, a function encapsulates a few lines of code. This associates a name with statements and allows us to reuse the code.

For example let's say we wanted to write some functions for drawing shapes:

```
# psuedo code
def draw_square():
    for i in range(4):
        go_forward(100) # value in px
        turn_left(90) # value in degrees
```

# Generalization

Generalization adds variables to functions so that the same function can be slightly altered.

```
# further generalize by adding an argument for length
def draw_square(length):
    for i in range(4):
        go_forward(length)
        turn_left(90)
```

# more generalization of the function:

We can make a polygon function.

# more generalization of the function:

We can make a polygon function.

Draw a pentagon

```
angle = 360 / 5
for i in range(5):
    go_forward(100)
    turn_left(angle)
```

# more generalization of the function:

We can make a polygon function.

Draw a pentagon

```
angle = 360 / 5
for i in range(5):
    go_forward(100)
    turn_left(angle)
```

Draw a hexagon

```
angle = 360 / 6
for i in range(6):
    go_forward(100)
    turn_left(angle)
```

# more generalization of the function:

We can make a polygon function.

Draw a pentagon

```
angle = 360 / 5
for i in range(5):
    go_forward(100)
    turn_left(angle)
```

Draw a hexagon

```
angle = 360 / 6
for i in range(6):
    go_forward(100)
    turn_left(angle)
```

After creating the code for a pentagon and hexagon, we can generalize to an n sided polygon:

```
def polygon(t, n, length):
    angle = 360 / n
    for i in range(n):
        go_forward(length)
        turn_left(angle)
```

# Lists

We will start with lists in Python

# Lists

We will start with lists in Python

## List Creation

Use square brackets. Lists can contain any mix of data types. You can nest lists inside other lists.

# Lists

We will start with lists in Python

## List Creation

Use square brackets. Lists can contain any mix of data types. You can nest lists inside other lists.

In [51]:
```
fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]
```

# Lists

We will start with lists in Python

## List Creation

Use square brackets. Lists can contain any mix of data types. You can nest lists inside other lists.

In [51]:

```
fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]
```

In [52]:

```
fam2 = [["liz", 1.73],
["emma", 1.68],
["mom", 1.71],
["dad", 1.89]]
```

# Lists

We will start with lists in Python

## List Creation

Use square brackets. Lists can contain any mix of data types. You can nest lists inside other lists.

In [51]:
```
fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]
```

In [52]:
```
fam2 = [["liz", 1.73],
["emma", 1.68],
["mom", 1.71],
["dad", 1.89]]
```

In [53]:
```
fam
```

Out[53]:
```
['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
```

# Lists

We will start with lists in Python

## List Creation

Use square brackets. Lists can contain any mix of data types. You can nest lists inside other lists.

In [51]:
```
fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]
```

In [52]:
```
fam2 = [["liz", 1.73],
["emma", 1.68],
["mom", 1.71],
["dad", 1.89]]
```

In [53]:
```
fam
```

Out[53]:
```
['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
```

In [54]:
```
fam2
```

Out[54]:
```
[['liz', 1.73], ['emma', 1.68], ['mom', 1.71], ['dad', 1.89]]
```

# Subsetting lists

- index starts at 0 (hardest part to adapt for R users)
- use a series of square brackets for nested lists
- use negative numbers to count from the end

# Subsetting lists

- index starts at 0 (hardest part to adapt for R users)
- use a series of square brackets for nested lists
- use negative numbers to count from the end

In [55]:

```
fam[0]
```

Out[55]:

```
'liz'
```

# Subsetting lists

- index starts at 0 (hardest part to adapt for R users)
- use a series of square brackets for nested lists
- use negative numbers to count from the end

In [55]:
```
fam[0]
```

Out[55]:
```
'liz'
```

In [56]:
```
fam2[0]
```

Out[56]:
```
['liz', 1.73]
```

# Subsetting lists

- index starts at 0 (hardest part to adapt for R users)
- use a series of square brackets for nested lists
- use negative numbers to count from the end

In [55]:
```
fam[0]
```

Out[55]:
```
'liz'
```

In [56]:
```
fam2[0]
```

Out[56]:
```
['liz', 1.73]
```

In [57]:
```
fam2[0][0]
```

Out[57]:
```
'liz'
```

In [58]:

```
fam[-1]
```

Out[58]:

```
1.89
```

In [58]:

```
fam[-1]
```

Out[58]:

```
1.89
```

In [59]:

```
fam2[-1]
```

Out[59]:

```
['dad', 1.89]
```

In [58]:
```
fam[-1]
```
Out[58]:
```
1.89
```
In [59]:
```
fam2[-1]
```
Out[59]:
```
['dad', 1.89]
```
In [60]:
```
fam2[-1][-1]
```
Out[60]:
```
1.89
```

# List Slicing

Note that the slice will not include the item in the index after the colon. You can think of the 'slice' happening at the commas corresponding to the number. So fam[1:3] slices the list at the first and third commas, and extracts [1.73, 'emma']

# List Slicing

Note that the slice will not include the item in the index after the colon. You can think of the 'slice' happening at the commas corresponding to the number. So fam[1:3] slices the list at the first and third commas, and extracts [1.73, 'emma']

In [61]:

```
fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]
fam[1:3]
```

Out[61]:

```
[1.73, 'emma']
```

# List Slicing

Note that the slice will not include the item in the index after the colon. You can think of the 'slice' happening at the commas corresponding to the number. So fam[1:3] slices the list at the first and third commas, and extracts [1.73, 'emma']

In [61]:

```
fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]
fam[1:3]
```

Out[61]:

```
[1.73, 'emma']
```

In [62]:

```
fam[1:2]
```

Out[62]:

```
[1.73]
```

# List Slicing

Note that the slice will not include the item in the index after the colon. You can think of the 'slice' happening at the commas corresponding to the number. So fam[1:3] slices the list at the first and third commas, and extracts [1.73, 'emma']

In [61]:
```
fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]
fam[1:3]
```

Out[61]:
```
[1.73, 'emma']
```

In [62]:
```
fam[1:2]
```

Out[62]:
```
[1.73]
```

In [63]:
```
fam[1]
```

Out[63]:
```
1.73
```

# List Slicing

Note that the slice will not include the item in the index after the colon. You can think of the 'slice' happening at the commas corresponding to the number. So fam[1:3] slices the list at the first and third commas, and extracts [1.73, 'emma']

In [61]:
```
fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]
fam[1:3]
```

Out[61]:
```
[1.73, 'emma']
```

In [62]:
```
fam[1:2]
```

Out[62]:
```
[1.73]
```

In [63]:
```
fam[1]
```

Out[63]:
```
1.73
```

In [64]:
```
fam[1:1]  # there is nothing between the first and first commas
```

Out[64]:
```
[]
```

In [65]:

```
fam[0:2]
```

Out[65]:

```
['liz', 1.73]
```

In [65]:

```
fam[0:2]
```

Out[65]:

```
['liz', 1.73]
```

In [66]:

```
fam[6:8]
```

Out[66]:

```
['dad', 1.89]
```

In [65]:
```
fam[0:2]
```
Out[65]: ['liz', 1.73]

In [66]:
```
fam[6:8]
```
Out[66]: ['dad', 1.89]

In [67]:
```
fam[2:]
```
Out[67]: ['emma', 1.68, 'mom', 1.71, 'dad', 1.89]

In [65]:
```
fam[0:2]
```
Out[65]:
```
['liz', 1.73]
```
In [66]:
```
fam[6:8]
```
Out[66]:
```
['dad', 1.89]
```
In [67]:
```
fam[2:]
```
Out[67]:
```
['emma', 1.68, 'mom', 1.71, 'dad', 1.89]
```
In [68]:
```
fam[:4]
```
Out[68]:
```
['liz', 1.73, 'emma', 1.68]
```

In [69]:
```python
fam[:]  # slice with no indices will create a (shallow) copy of the list.
```

Out[69]:
```
['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
```

In [69]:

```python
fam[:]  # slice with no indices will create a (shallow) copy of the list.
```

Out[69]:

```
['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
```

In [70]:

```python
fam[] # throws error
```

```
  File "<ipython-input-70-792e48a646bd>", line 1
    fam[] # throws error
        ^
SyntaxError: invalid syntax
```

In [69]:

```
fam[:]  # slice with no indices will create a (shallow) copy of the list.
```

Out[69]:

```
['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
```

In [70]:

```
fam[] # throws error
```

```
  File "<ipython-input-70-792e48a646bd>", line 1
    fam[] # throws error
        ^
SyntaxError: invalid syntax
```

In [71]:

```
fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]
print(fam)
print(fam[-5:-2])
```

```
['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
[1.68, 'mom', 1.71]
```

In [72]:

```
fam2
```

Out[72]:

```
[['liz', 1.73], ['emma', 1.68], ['mom', 1.71], ['dad', 1.89]]
```

In [73]:

```
fam2[1:3]
```

Out[73]:

```
[['emma', 1.68], ['mom', 1.71]]
```

In [74]:

```
fam2[1:3][0][0:1]
```

Out[74]: ['emma']

# Lists are mutable

This means that methods change the lists themselves. If the list is assigned to another name, both names refer to the exact same object.

# Lists are mutable

This means that methods change the lists themselves. If the list is assigned to another name, both names refer to the exact same object.

In [75]:

```
fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]
print(fam)
second = fam    # second references fam. second is not a copy of fam.
second[0] = "sister"  # we make a change to the list 'second'
print(second)
print(fam) # changing the list 'second' has changed the list 'fam'
```

```
['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
['sister', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
['sister', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
```

In [76]:

```python
fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]
print(fam)
second = fam[:]  # creates a copy of the list
# second = fam.copy() # you can also create a list using the copy() method
second[0] = "sister"
print(second)
print(fam) # changing the list second does not modify fam because second is a copy
```

```
['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
['sister', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
```

In [77]:
```
third = fam.copy()
print(third)
third[1] = 1.65
print(third)
print(fam)
```

```
['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
['liz', 1.65, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
```

In [78]:
```
fam
```

Out[78]:
```
['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
```

In [79]:
```
list2 = list(fam)
```

In [80]:
```
list2[1] = 1.9
```

In [81]:
```
list2
```

Out[81]:
```
['liz', 1.9, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
```

In [82]:
```
fam
```

Out[82]:
```
['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
```

You can use list slicing in conjuction with assignment to change values

You can use list slicing in conjuction with assignment to change values

In [83]:

```
print(fam)
fam[1:3] = [1.8, "jenny"]
print(fam)
```

```
['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
['liz', 1.8, 'jenny', 1.68, 'mom', 1.71, 'dad', 1.89]
```