Lecture 5 - Conditionals and Recursion

Week 3 Wednesday

Miles Chen, PhD

Adapted from Chapter 5 of Think Python by Allen B Downey

List content adapted from "Whirlwind Tour of Python" by Jake VanderPlas

Quick Review of some operators:

Floor division and modulus

In [1]:

```
# regular division
minutes = 105
minutes / 60
```

Out[1]:

```
1.75
```

In [2]:

```
# floor division
minutes = 105
hours = minutes // 60
hours
```

Out[2]:

```
1
```

In [3]:

```
# modulus
remainder = minutes % 60
remainder
```

Out[3]:

```
45
```

You can check if something is even by using modulus 2 == 0

In [4]:

```
# check if something is even
x = 5
x % 2 == 0
```

Out[4]:

```
False
```

In [5]:

```
# check if something is even
x = 6
x % 2 == 0
```

Out[5]:

```
True
```

Boolean expressions

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

```
Out[6]: True
```

```
In [7]: 5 == 6
```

```
Out[7]: False
```

```
In [8]: 5 != 6
```

```
Out[8]: True
```

```
In [9]:  5 > 6

Out[9]:  False

In [10]: 5 < 6

Out[10]: True

In [11]: 5 >= 6

Out[11]: False

In [12]: 5 <= 6

Out[12]: True
```

Logical operators

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

```
In [13]: True and True

Out[13]: True

In [14]: True and False

Out[14]: False

In [15]: True or False

Out[15]: True

In [16]: not True

Out[16]: False

In [17]: not False

Out[17]: True
```

```
In [18]: False or not False
```

```
Out[18]: True
```

```
In [19]: True and not False
```

```
Out[19]: True
```

```
In [20]: n = 6
         n % 2 == 0 and n % 3 == 0
```

```
Out[20]: True
```

```
In [21]: n = 8
         n % 2 == 0 and n % 3 == 0
```

```
Out[21]: False
```

```
In [22]: n = 8
         n % 2 == 0 or n % 3 == 0
```

```
Out[22]: True
```

Conditionals

if statements start with if followed by a logical expression that is either true or false, and then a colon.

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.

For a single logical expression, parentheses are not required. For more complex logical expressions, you may need to use parentheses.

In [23]:

```
x = 5
if x > 0:
    print('x is positive')
```

```
x is positive
```

The `pass` statement does nothing. You can enter it in a place where there should be code, but haven't figured out what to write yet.

In [24]:

```
x = -3
if x < 0:
    pass
```

else

The `else` line is written on the same indent level as the `if` statement. Indentation indicate which lines are associated with the `else`. The else statement is evaluated only if the expression in the if statement is `False`

In [27]:

```python
x = 7
if x % 2 == 0:
    print('x is even')
else:
    print('x is odd')
```

```
x is odd
```

elif

In [28]:

```
x = -2
if x > 0:
    print('x is positive')
elif x < 0:
    print('x is negative')
else:
    print('x is zero')
```

```
x is negative
```

Like `else` statements, `elif` statements will only be evaluated if the expression in the `if` statement is `False`

In [29]:

```
x = 5
if x > 0:
    print('x is positive')
elif x > 3: # will never be true when the if is False
    print('x is greater than 3')
elif x < 0:
    print('x is negative')
else:
    print('x is zero')
```

```
x is positive
```

In [31]:

```
x = 5
if x > 0:
    print('x is positive')
if x > 3:
    print('x is greater than 3')
if x < 0:
    print('x is negative')
if x == 0:
    print('x is zero')
```

```
x is positive
x is greater than 3
```

Nested Conditionals

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

In [32]:

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

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

In [33]:

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

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

In [34]:

```
# 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.

In [37]:

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

In [40]:

```
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 [41]:

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

In [42]:

```
def print_n(s, n):
    if n <= 0:
        return # exits the function
    print(s)
    print_n(s, n - 1)
```

In [43]:

```
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

In [44]:

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

In [45]:

```
factorial(4)
```

Out[45]:

```
24
```

In [46]:

```
factorial(5)
```

Out[46]:

```
120
```

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

```
In [48]: fam2 = [["liz", 1.73],
         ["emma", 1.68],
         ["mom", 1.71],
         ["dad", 1.89]]
```

```
In [49]: fam

Out[49]: ['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
```

```
In [50]: fam2

Out[50]: [['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

```
In [51]: fam[0]

Out[51]: 'liz'

In [52]: fam2[0]

Out[52]: ['liz', 1.73]

In [53]: fam2[0][0]

Out[53]: 'liz'
```

```
In [54]: fam[-1]

Out[54]: 1.89

In [55]: fam2[-1]

Out[55]: ['dad', 1.89]

In [56]: fam2[-1][-1]

Out[56]: 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']

```
In [57]: fam = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]
         fam[1:3]
```

```
Out[57]: [1.73, 'emma']
```

```
In [58]: fam[1:2]
```

```
Out[58]: [1.73]
```

```
In [59]: fam[1]
```

```
Out[59]: 1.73
```

```
In [60]: fam[1:1]   # there is nothing between the first and first commas
```

```
Out[60]: []
```

```
In [61]: fam[0:2]

Out[61]: ['liz', 1.73]

In [62]: fam[6:8]

Out[62]: ['dad', 1.89]

In [63]: fam[2:]

Out[63]: ['emma', 1.68, 'mom', 1.71, 'dad', 1.89]

In [64]: fam[:4]

Out[64]: ['liz', 1.73, 'emma', 1.68]
```

```
In [65]:  fam[:]  # slice with no indices will create a (shallow) copy of the list.

Out[65]:  ['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
```

```
In [66]:  fam[] # throws error

            File "<ipython-input-66-792e48a646bd>", line 1
              fam[] # throws error
                  ^
          SyntaxError: invalid syntax
```

```
In [67]:  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 [68]:  fam2

Out[68]:  [['liz', 1.73], ['emma', 1.68], ['mom', 1.71], ['dad', 1.89]]
```

```
In [69]:  fam2[1:3]

Out[69]:  [['emma', 1.68], ['mom', 1.71]]
```

```
In [72]:  fam2[1:3][0][0:1]

Out[72]:  ['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.

In [73]:

```
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 [74]:

```
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 [75]: 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 [77]: fam

Out[77]: ['liz', 1.8, 'jenny', 1.68, 'mom', 1.71, 'dad', 1.89]
```

```
In [78]: list2 = list(fam)
```

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

```
In [81]: list2

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

```
In [82]: fam

Out[82]: ['liz', 1.8, 'jenny', 1.68, 'mom', 1.71, 'dad', 1.89]
```

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

In [76]:

```
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]
```