Lecture 4: Booleans and Conditionals*

Boolean Values

Python has two values of the `bool` type, written `True` and `False`. These are called logical values or **Boolean values**, named after 19th century mathematician George Boole.

[104]:
```
True
```

[104]:
```
True
```

[105]:
```
type(True)
```

[105]:
```
bool
```

[106]:
```
False
```

[106]:
```
False
```

[107]:
```
type(False)
```

[107]:
```
bool
```

Careful, the two values are written as uppercase, and you'll get an error if misspelled:

[108]:
```
true
```

```
NameError: name 'true' is not defined
```

[109]:
```
false
```

```
NameError: name 'false' is not defined
```

Relational Operators

We have seen arithmetic operators that produce as output numerical values. Today, we'll see **relational operators** that
produce as output **Boolean values**. Relational operators are used to compare two values.

[110]:
```
3 < 5
```

***Acknowledgement.** This notebook has been adapted from the Wellesley CS111 Spring 2019 course materials (http://cs111.wellesley.edu/spring19).

[110]: True

[111]:
```
3 > 2
```

[111]: True

[112]:
```
5 == 5
```

[112]: True

[113]:
```
5 >= 5
```

[113]: True

[114]:
```
6 <= 5
```

[114]: False

Note: == is pronounced *"equals"* and != is pronounced *"not equals"*. This is why we distinguish the pronunciation
of the single equal sign = as "gets", which is assignment and nothing to do with mathematical equality!

Relational operators can also be used to compare strings (in dictionary order).

[115]:
```
'bat' < 'cat'
```

[115]: True

[116]:
```
'bat' < 'ant'
```

[116]: False

[117]:
```
'bat' == 'bat'
```

[117]: True

[118]:
```
'bat' < 'bath'
```

[118]: True

[119]:
```
'Cat' < 'bat'
```

[119]: True

EXPLANATION: How does this comparison of string values work? Python starts by comparing the first character of each string to one another. For example "b" with "c". Because the computer doesn't know anything about letters, it converts everything into numbers. Each character has a

numerical code that is summarized in this table of ASCII codes. In Python, we can look up the ASCII code via the Python built-in function `ord`:

[120]:
```
print(ord('a'), ord('b'), ord('c'))
```

```
97 98 99
```

As you can see, the value for 'b', 98, is smaller than the value for 'c', 99, thus, `'b' < 'c'`. Once two unequal characters are found, Python stops comparing the other characters, because there is no point in continuing. However, if characters are the same, like in 'bat' and 'bath', the comparisons continue until the point in which something that differs is found. In this case, there is an extra 't', making 'bath' greater in value than 'bat'.

Uppercase vs. Lowercase: Counterintuitively, it turns out, the upppercase letters are internally represented with smaller numbers than lowercase letters. See the ASCII table and the examples below:

[121]:
```
print(ord('A'), ord('a'))
```

```
65 97
```

[122]:
```
print(ord('B'), ord('b'))
```

```
66 98
```

This explains why the world 'Cat' is smaller than the word 'cat'.

Logical Operators

There are three logical operators: `not`, `and`, `or`, which are applied on expressions that are already evaluated as boolean values.

`not`

not *expression* evaluates to the opposite of the truth value of *expression*

[123]:
```
not (3 > 5)
```

[123]:
```
True
```

[124]:
```
not (3 == 3)
```

[124]:
```
False
```

`and`

exp1 **and** *exp2* evaluates to `True` iff **both** *exp1* and *exp2* evaluate to `True`.

[125]:
```
True and True
```

[125]: True

```
[126]: True and False
```

[126]: False

```
[127]: (3 < 5) and ('bat' < 'ant')
```

[127]: False

```
[128]: (3 < 5) and ('bat' < 'cat')
```

[128]: True

or

exp1 **or** *exp2* evaluates to `True` iff **at least one** of *exp1* and *exp2* evaluate to `True`.

```
[129]: True or True
```

[129]: True

```
[130]: True or False
```

[130]: True

```
[131]: (3 > 5) or ('bat' < 'cat')
```

[131]: True

```
[132]: (3 > 5) or ('bat' < 'ant')
```

[132]: False

Membership Operator `in` for Sequences

Let us practice some predicates and conditionals that involve **sequences**. A *sequence* is an ordered collection of items. For example, a `string` is a sequence as it is a just an ordered collection of letters. A `list` in python is a another sequence, it is a special data type which stores an ordered collection of items.

We will cover strings and lists in more detail in the coming lectures. Today we will use the `in` operator and `not in` operators to create Boolean expressions involving sequences.

`in` **operator:** `s1 in s2` tests if string `s1` is a substring of string `s2`
`not in` **operator:** returns the opposite of `in`, i.e., `s1 not in s2` is the same as `not s1 in s2`

```
[133]: '134' in 'CS134'
```

[133]:
```
True
```

[134]:
```
'era' not in 'generation'
```

[134]:
```
False
```

[135]:
```
'grass' not in 'grassroots'
```

[135]:
```
False
```

Lists and the `in` operator: A list in python is just a collection of values enclosed in `[]`. `item in myList` tests if `item` is present in the list `myList`.

[136]:
```
evenNums = [2, 4, 6, 8, 10]  # list of even numbers less than equal to 10
```

[137]:
```
4 in evenNums
```

[137]:
```
True
```

[138]:
```
5 in evenNums
```

[138]:
```
False
```

[139]:
```
nameList = ['Anna', 'Chris', 'Zoya', 'Sherod', 'Zack']
```

[140]:
```
'Shikha' in nameList
```

[140]:
```
False
```

[141]:
```
'Chris' in nameList
```

[141]:
```
True
```

Predicates

Definition: A predicate is simply any function that returns a boolean value.

Usually, the function body will contain a complex expression combining relational and logical expressions, as the following examples show:

[142]:
```
def isHogwartsHouse(s):
    '''Given a string, returns True if it is a Hogwarts house'''
    return (s == 'Gryffindor' or s == 'Hufflepuff'
            or s == 'Ravenclaw' or s == 'Slytherin')
    # notice the use of parenthesis to enclose multi-line expressions
```

[143]:
```
isHogwartsHouse('Slytherin')
```

[143]: True

[144]:
```
isHogwartsHouse("Hagrid's hut")
```

[144]: False

Expressing intervals of numbers: We can combine relational expressions to create intervals of numbers that fulfill certain criteria. Below is a predicate that checks if a value is within a given interval of numbers.

[145]:
```
def isBetween(n, lo, hi):
    """determines if n is between lo and hi"""
    return (lo <= n) and (n <= hi)
```

More fun with Math: Is a number divisible by a factor? Is it even?

[146]:
```
def isDivisibleBy(num, factor):
    return (num % factor) == 0 # notice the remainder operator
```

[147]:
```
isDivisibleBy(121, 11)
```

[147]: True

[148]:
```
isDivisibleBy(25, 3)
```

[148]: False

Strings and Predicates Let us practice some predicates and conditionals that involve strings.

Write the predicate `isVowel` that takes a character as input and returns true if it is a vowel.

Helper function lower(): `word.lower()` returns a new string which is the string `word` in all lowercase letters.

[205]:
```
# define the isVowel function

def isVowel1(char):
    '''Takes a char as input and determines if it is a vowel.
    Function version without in'''
    c = char.lower()
    return c == 'a' or c == 'e' or c == 'i' or \
            c == 'e' or c == 'o' or c == 'u'
    # can also use \ for multi-line expressions

def isVowel2(char):
    '''Takes a char as input and determines if it is a vowel.
    Function version using in'''
    c = char.lower()
    return c in 'aeiou'
```

[154]:
```
help(isVowel1)   # calling help on a function returns its docstring
```

```
Help on function isVowel1 in module __main__:

isVowel1(char)
    Takes a char as input and determines if it is a vowel.
    Function version without in
```

[155]:
```
print(isVowel1('e'), isVowel2('e'))
```

```
True True
```

[156]:
```
print(isVowel1('b'), isVowel2('b'))
```

```
False False
```

[157]:
```
print(isVowel1('U'), isVowel2('U'))
```

```
True True
```

Simple conditionals: If Statements

An `if` statement (also called a **condtional** statement) chooses between two branches based on a test value.

[158]:
```
def abs(n):
    '''Return the absolute value of the number n'''
    if n >= 0:
        return n
    else:
        return -n

def classify(num):
    '''Return a string indicating whether num is negative or not.'''
    if num < 0:
        return 'negative'
    else:
        return 'nonnegative'
```

[159]:
```
abs(-17)
```

[159]: 17

[160]:
```
abs(111)
```

[160]:
```
111
```

[161]:
```
classify(-17)
```

[161]:
```
'negative'
```

[162]:
```
classify(111)
```

[162]:
```
'nonnegative'
```

A function with a conditional might print something.

[163]:
```
def doWhenTemperature(temp):
    if temp <= 65:
        print("Put on a sweater or coat.")
    else:
        print("You can wear short sleeves today.")
```

[164]:
```
doWhenTemperature(72)
```

```
You can wear short sleeves today.
```

[165]:
```
doWhenTemperature(50)
```

```
Put on a sweater or coat.
```

Does `doWhenTemperature` return anything?

[166]:
```
print(doWhenTemperature(50))
```

```
Put on a sweater or coat.
None
```

Function bodies and conditional branches with multiple statements

[167]:
```
def categorize(num):
    '''This function has 3 statements in its body.
       They are executed from top to bottom, one after the other.
    '''
    print('Categorizing', num)
    if num % 2 == 0:
        print("It's even")
    else:
        print("It's odd")
    if num < 0:
        '''This branch has 2 statements.'''
        print("It's negative")
        print("(That means it's less than zero)")
```

```
    else:
        print("It's nonnegative")
```

[168]:
```
categorize(111)
```

```
Categorizing 111
It's odd
It's nonnegative
```

[169]:
```
categorize(-20)
```

```
Categorizing -20
It's even
It's negative
(That means it's less than zero)
```

The `pass` statement and dropping `else`

When we don't want to do anything in a conditional branch, we use the special `pass` statement, which means "do nothing". (It's a syntax error to leave a branch blank.)

[170]:
```
def warnWhenTooFast(speed):
    if speed > 55:
        print("Slow down! You're going too fast")
    else:
        pass # do nothing
```

[171]:
```
warnWhenTooFast(75)
```

```
Slow down! You're going too fast
```

[172]:
```
warnWhenTooFast(40)
```

It's OK to have an `if` statement without an `else` clause. In this case, the missing `else` clause is treated as if it were a `pass` statement.

[173]:
```
def warnWhenTooFast2(speed):
    if speed > 55:
        print("Slow down! You're going too fast")
```

[174]:
```
warnWhenTooFast2(75)
```

```
Slow down! You're going too fast
```

[175]:
```
warnWhenTooFast2(40)
```

Below are two correct variants of the `abs` absolute value function defined above. Explain why they work.

[176]:
```
def abs2(n):
    '''returns the absolute value of n'''
    result = n
    if n < 0:
        result = -n
    return result

print(abs2(-17), abs2(42))
```

```
17 42
```

[177]:
```
def abs3(n):
    '''returns the absolute value of n'''
    if n < 0:
        return -n
    return n

print(abs3(-17), abs3(42))
```

```
17 42
```

Nested and chained conditionals

It often make sense to have a conditional statement nested inside the branch of another conditional.

Below we show variants of a function that returns the movie rating appropriate for a given age of movier goer. (If you want to learn more about film ratings, read this Wikipedia article.)

[178]:
```
def movieAge1(age):
    if age < 8:
        return 'G'
    else:
        if age < 13:
            return 'PG'
        else:
            if age < 18:
                return 'PG-13'
            else:
                return 'R'
```

[179]:
```
movieAge1(5)
```

[179]:
```
'G'
```

[180]:
```
movieAge1(10)
```

[180]:
```
'PG'
```

[181]:
```
movieAge1(15)
```

[181]:
```
'PG-13'
```

[182]:
```
movieAge1(20)
```

[182]:
```
'R'
```

Python uses **chained** (**multibranch**) conditionals with `if`, `elifs`, and `else` to execute exactly one of several branches.

[183]:
```
def movieAge2(age):
    if age < 8:
        print('G')
    elif age < 13:
        print('PG')
    elif age < 18:
        print('PG-13')
    else:
        print('R')
```

[184]:
```
movieAge2(5)
```

```
G
```

[185]:
```
movieAge2(10)
```

```
PG
```

[186]:
```
movieAge2(15)
```

```
PG-13
```

[187]:
```
movieAge2(20)
```

```
R
```

Remember: Only the **first** true branch will be executed.

Important: As shown in the following example, the order of chaining conditionals matters!

[188]:
```
def movieAgeWrong(age):
    if age < 18:
        print('PG-13')
    elif age < 13:
        print('PG')
    elif age < 8:
        print('G')
    else:
```

```
        print('R')
```

[189]:
```
movieAgeWrong(5)
```

```
PG-13
```

[190]:
```
movieAgeWrong(10)
```

```
PG-13
```

[191]:
```
movieAgeWrong(15)
```

```
PG-13
```

[192]:
```
movieAgeWrong(20)
```

```
R
```

Exercise: `daysInMonth`

Define a function named `daysInMonth` that takes a month (as an integer) as the argument, and returns the number of days in it, assuming the year is not a leap year.

Assume 1 is January, 2 is February, . . . , 12 is December. If the month does not fall between 1 and 12, return an error message as a string.

Make the function as concise as possible (group months by days, don't write 12 separate if-else clauses).

[193]:
```
# Define your daysInMonth function below

def daysInMonth(month):
    '''Given a month between 1-12, returns the number of days in it,
    assuming the year is not a leap year'''
    if month < 1 or month > 12:
        return 'Error: Month does not fall between 1-12'
    elif month == 2:
        return 28
    elif month == 4 or month == 6 == month == 9 or month == 11:
        return 30
    else:
        return 31
```

[194]:
```
daysInMonth(4)  # April
```

[194]:
```
30
```

[195]:
```
daysInMonth(8)  # August
```

[195]: 31

[196]:
```
daysInMonth(2)  # February
```

[196]: 28

[197]:
```
daysInMonth(13) # Error message
```

[197]: 'Error: Month does not fall between 1-12'

[ ]:

Improving the style of functions with conditionals

Having seen conditional statements, you may be tempted to use them in predicates. But most predicates can be defined without conditionals by using combinations of relational and logical operators. For example, compare the complicated and simplifed functions below:

[198]:
```
def isFreezingComplex(temp):
    if temp <= 32:
        return True
    else:
        return False

def isFreezingSimple(temp):
    return temp <= 32
```

[199]:
```
print(isFreezingComplex(20), isFreezingSimplified(20))
```

```
True True
```

[200]:
```
print(isFreezingComplex(72), isFreezingSimplified(72))
```

```
False False
```

[201]:
```
def isPositiveEvenComplex(num):
    if num > 0:
        if num % 2 == 0:
            return True
        return False
    return False

def isPositiveEvenSimple(num):
    return num > 0 and num % 2 == 0
```

[202]:
```
print(isPositiveEvenComplex(12), isPositiveEvenSimple(12))
```

```
True True
```

[203]:
```
print(isPositiveEvenComplex(19), isPositiveEvenSimple(19))
```

```
False False
```

[204]:
```
print(isPositiveEvenComplex(-3), isPositiveEvenSimple(-3))
```

```
False False
```