In [1]:
```
# Styling notebook
from IPython.core.display import HTML
def css_styling():
    styles = open("./styles/custom.css", "r").read()
    return HTML(styles)
css_styling()
```

Out[1]:

Sets and applications

Python has a pretty decent implementation of sets, of course via hash tables. They're pretty good to demonstrate the basic set operations

- test for empty set
- element
- subset
- equality
- union
- intersection
- relative complement: $A \backslash B = \{x \in A \mid x \notin B\}$
- symmetric difference: $A \Delta B = (A \backslash B) \cup (B \backslash A)$

In [13]:
```
# test for empty set
def is_empty(a) :
    return not a

# test for element
def isElement(elem, aSet) :     # pretty much built in
    return elem in aSet

# test for subset
def isSubset(set1, set2) :
    for elem in set1 :          # test all elements in set1
        if not elem in set2 :   # if they're in set2
            return False
    return True

# test for equality
def setEqual(set1, set2) :
    return isSubset(set1,set2) and isSubset(set2,set1)

def intersection(set1, set2) :
    # return set1.intersection(set2)   # also built-in
    # this is the "pedestrian" solution:
    result = set()
    for elem in set1 :          # Check every element of set1,
        if elem in set2 :       # if it's in set1,
            result.add(elem)    # it goes into the intersection
    return result

def unionWrong(set1, set2) :
    # This one contains a subtle bug...
    result = set1           # Start with set1
    for elem in set2 :      # Add every element of set2
        result.add(elem)
    return result

def union(set1, set2) :
    # This is fine
    result = set1.copy()    # Start with A COPY OF set1
    for elem in set2 :      # Add every element of set2
        result.add(elem)
    return result

def relComplement(set1, set2) :
    result = set()
    for elem in set1 :          # Check every element of set1,
        if not elem in set2 :   # if it's NOT in set1,
            result.add(elem)    # it goes into the relative complement
    return result

def symmDiff(set1, set2) :
    return union(relComplement(set1,set2),relComplement(set2,set1))

empty = set()
set1 = {6, 4, 2, 7, 9, 1}
set2 = {7, 4, 9}
set3 = {6, 12, 9}
```

```
print("3 in {6, 4, 2, 7, 9, 1}:",isElement(3,set1))
print("{6, 12, 9} subset {6, 4, 2, 7, 9, 1}:",isSubset(set3,set1))
print("{} subset {6, 12, 9}:",isSubset(empty,set3))
print("{7, 4, 9} == {6, 12, 9}:",setEqual(set2,set3))
print("{6, 4, 2, 7, 9, 1} == {6, 4, 2, 7, 9, 1}:",setEqual(set1,set1))
print("{7, 4, 9} intersect {6, 12, 9} =",intersection(set2,set3))
# print(unionWrong(set2, set3))
# print(set2)
print("{7, 4, 9} union {6, 12, 9} =",union(set2,set3))
print("{6, 4, 2, 7, 9, 1} \ {6, 12, 9} =",relComplement(set1,set3))
print("{6, 4, 2, 7, 9, 1} DELTA {6, 12, 9} =",symmDiff(set1,set3))
```

```
3 in {6, 4, 2, 7, 9, 1}: False
{6, 12, 9} subset {6, 4, 2, 7, 9, 1}: False
{} subset {6, 12, 9}: True
{7, 4, 9} == {6, 12, 9}: False
{6, 4, 2, 7, 9, 1} == {6, 4, 2, 7, 9, 1}: True
{7, 4, 9} intersect {6, 12, 9} = {9}
{7, 4, 9} union {6, 12, 9} = {4, 6, 7, 9, 12}
{6, 4, 2, 7, 9, 1} \ {6, 12, 9} = {1, 2, 4, 7}
{6, 4, 2, 7, 9, 1} DELTA {6, 12, 9} = {1, 2, 4, 7, 12}
```

But sets aren't hashable themselves! So there are no such things as sets of sets in Python.

That's particularly weird if you want to do power sets. So we need a workaround over lists/tuples or something like that.

Recursion schema for power set construction:

1. $2^{\emptyset} = \{\emptyset\}$
2. To construct $2^{\{a,b,c,\ldots\}}$ construct recursively $2^{\{b,c,\ldots\}}$. Then every $s \in 2^{\{b,c,\ldots\}}$ is also $\in 2^{\{a,b,c,\ldots\}}$, but also every $a \cup s$ where $s \in 2^{\{b,c,\ldots\}}$

The only tricky thing is to make sure we understand which operations are destructive and which aren't. If they are destructive, make a copy before applying them so you don't change the original!

In [3]:
```
def powerSet(s) :                  # Now we let this be and return a LIST
                                   # of the subsets of s - as close as we can get
    if is_empty(s) : return [[]]

    sCopy = s.copy()
    firstS = sCopy[0]                # First element of the list
    sCopy.remove(firstS)             # The rest

    recursivePS = powerSet(sCopy)   # YAY! Finally recursion
    result = []
    for subset in recursivePS :
        result.append(subset)
        withFirst = [ firstS ] + subset.copy()
        result.append(withFirst)
    return result


def cartProd(s1,s2) :
    result = []
    for a in s1 :
        for b in s2 :
            result.append([a,b])
    return result
```

In [4]:
```
print(powerSet(['a','b','c']))
print(cartProd(['a','b','c'],[0,1]))
```

```
[[], ['a'], ['b'], ['a', 'b'], ['c'], ['a', 'c'], ['b', 'c'], ['a', 'b', 'c']]
[['a', 0], ['a', 1], ['b', 0], ['b', 1], ['c', 0], ['c', 1]]
```

Fun with power sets: The Knapsack problem

The problem is pretty simple: Given a collection of "items" each with a weight and a value a thief wants to pick items that a) have the maximum value and b) don't exceed a maximum weight (the thief has a maximum weight capacity and might not be able to carry all items.

The thief has to make a decision for each item: Take it or leave it.

We call a set of items *admissible* if the total weigth does not exceed the maximum weight

A primitive solution is easy to construct:

1. Construct the power set of all items
2. Check each of the subsets if it's admissible
3. From the admissible ones, pick the one with the highest value

Of course this is a terribly inefficient solution, since for n items we would have to check 2^n (exponentially many) subsets!

More about that in your "Algorithms and Data Structures" class or in Wikipedia: Knapsack Problem (https://en.wikipedia.org/wiki/Knapsack_problem)

In [5]:
```
items = [['lamp',5,100], ['vase',3,2000], ['book', 4, 800], ["skeleton", 7, 10]]
powerSet(items)
```

Out[5]:
```
[[],
 [['lamp', 5, 100]],
 [['vase', 3, 2000]],
 [['lamp', 5, 100], ['vase', 3, 2000]],
 [['book', 4, 800]],
 [['lamp', 5, 100], ['book', 4, 800]],
 [['vase', 3, 2000], ['book', 4, 800]],
 [['lamp', 5, 100], ['vase', 3, 2000], ['book', 4, 800]],
 [['skeleton', 7, 10]],
 [['lamp', 5, 100], ['skeleton', 7, 10]],
 [['vase', 3, 2000], ['skeleton', 7, 10]],
 [['lamp', 5, 100], ['vase', 3, 2000], ['skeleton', 7, 10]],
 [['book', 4, 800], ['skeleton', 7, 10]],
 [['lamp', 5, 100], ['book', 4, 800], ['skeleton', 7, 10]],
 [['vase', 3, 2000], ['book', 4, 800], ['skeleton', 7, 10]],
 [['lamp', 5, 100], ['vase', 3, 2000], ['book', 4, 800], ['skeleton', 7, 10]]]
```

In [6]:
```
# Compute total value and total weight of a set of items
def eval(items) :
    value, weight = 0, 0;
    for item in items :
        weight = weight + item[1]
        value = value + item[2]
    return weight, value

def knapSackBruteForce(items, maxWeight) :
    bestValue = -1             # Comparison value
    bestSelection = []         # Best selection so far

    for selection in powerSet(items) :    # Run over all subsets
        # Find total value and weight of all items in this subset
        weight, value = eval(selection)

        if weight <= maxWeight and value > bestValue : # admissible and better??
            bestSelection = selection                  # Yep, store it!
            bestValue = value
    return bestSelection, eval(bestSelection)          # Best solution and its da
```

In [7]:
```
knapSackBruteForce(items,10)
```

Out[7]:
```
([['vase', 3, 2000], ['book', 4, 800]], (7, 2800))
```

Select subset via predicate

Given a set S and a one-place predicate (boolean function) P, select the subset $S \supseteq S' = \{x \in S | P(x)\}$

Note that in Python we can pass functions as parameters to other functions and just call them in this other function (see how P works in the example)

In [8]:
```
def select(S,P) :
    result = []
    for x in S :
        if P(x) : result.append(x)
    return result
```

In [9]:
```
def isOdd(n) :
    return n%2 == 1

aSet = [1,2,3,4,5,6]

print(select(aSet,isOdd))

# or (functional)
print(list(filter(isOdd,aSet)))

# or (list comprehension)
print([x for x in aSet if isOdd(x)])

# Yes, we could have implemented "select" via filter or via list comprehension, t
```

```
[1, 3, 5]
[1, 3, 5]
[1, 3, 5]
```

Fun with the Cartesian Product: Data combination

Assume database tables of the form $(Student, Course)$ and $(Course, Instructor)$

The Cartesian Product of these tables contains all pairs and is the first step in matching students and instructors via the course.

In [10]:
```
def printLines(aList) :
    for item in aList : print(item)

table1 = [['Jester', 'COT3100'], ['Beauregard', 'COT3100'], ['Caduceus', 'COT3100
          ['Jester', 'COP4027'], ['Caleb', 'COP4027'], ['Nott', 'COP4027']]

table2 = [['COT3100', 'Doctor OK'], ['COP4027', 'Doctor Coffey']]

cartProd(table1, table2)
```

Out[10]:
```
[[['Jester', 'COT3100'], ['COT3100', 'Doctor OK']],
 [['Jester', 'COT3100'], ['COP4027', 'Doctor Coffey']],
 [['Beauregard', 'COT3100'], ['COT3100', 'Doctor OK']],
 [['Beauregard', 'COT3100'], ['COP4027', 'Doctor Coffey']],
 [['Caduceus', 'COT3100'], ['COT3100', 'Doctor OK']],
 [['Caduceus', 'COT3100'], ['COP4027', 'Doctor Coffey']],
 [['Jester', 'COP4027'], ['COT3100', 'Doctor OK']],
 [['Jester', 'COP4027'], ['COP4027', 'Doctor Coffey']],
 [['Caleb', 'COP4027'], ['COT3100', 'Doctor OK']],
 [['Caleb', 'COP4027'], ['COP4027', 'Doctor Coffey']],
 [['Nott', 'COP4027'], ['COT3100', 'Doctor OK']],
 [['Nott', 'COP4027'], ['COP4027', 'Doctor Coffey']]]
```

Selecting specific pairs over the course

Here we just use a predicate that is T if the second element of the first tuple is equal to the first element of the second tuple. This will result in discarding all pairs where the courses don't match.

This is the basis of one of the simplest Relational DB operations: The "Equi-Join"

In [11]:
```
def match (pair) :
    # Second element of the first tuple == First element of the second tuple (cou
    return pair[0][1] == pair[1][0]

select(cartProd(table1,table2), match)
```

Out[11]:
```
[[['Jester', 'COT3100'], ['COT3100', 'Doctor OK']],
 [['Beauregard', 'COT3100'], ['COT3100', 'Doctor OK']],
 [['Caduceus', 'COT3100'], ['COT3100', 'Doctor OK']],
 [['Jester', 'COP4027'], ['COP4027', 'Doctor Coffey']],
 [['Caleb', 'COP4027'], ['COP4027', 'Doctor Coffey']],
 [['Nott', 'COP4027'], ['COP4027', 'Doctor Coffey']]]
```