Week 3

Sun Jun
with slides from Hans Petter Langtangen

Nested lists: list of lists

- A list can contain "any" object, also another list
- Instead of storing a table as two separate lists (one for each column), we can stick the two lists together in a new list:

```
Cdegrees = range(-20, 41, 5)
Fdegrees = [(9.0/5)*C + 32 for C in Cdegrees]

table1 = [Cdegrees, Fdegrees]  # list of two lists
```

- `table1[0]` is the `Cdegrees` list
- `table1[1]` is the `Fdegrees` list
- `table1[1][2]` is the 3rd element in `Fdegrees`

Table of coloumns vs table of rows

- The previous `table = [Cdegrees,Fdegrees]` is a table of (two) columns
- Let us make a table of rows instead, each row is a `[C,F]` pair:

```
table2 = []
for C, F in zip(Cdegrees, Fdegrees):
    row = [C, F]
    table2.append(row)

# more compact with list comprehension:
table2 = [[C, F] for C, F in zip(Cdegrees, Fdegrees)]
```

- `print table2` gives

```
[[-20, -4.0], [-15, 5.0], ......., [40, 104.0]]
```

- Iteration over a nested list:

```
for C, F in table2:
    # work with C and F from a row in table2
```

Illustration of table of columns

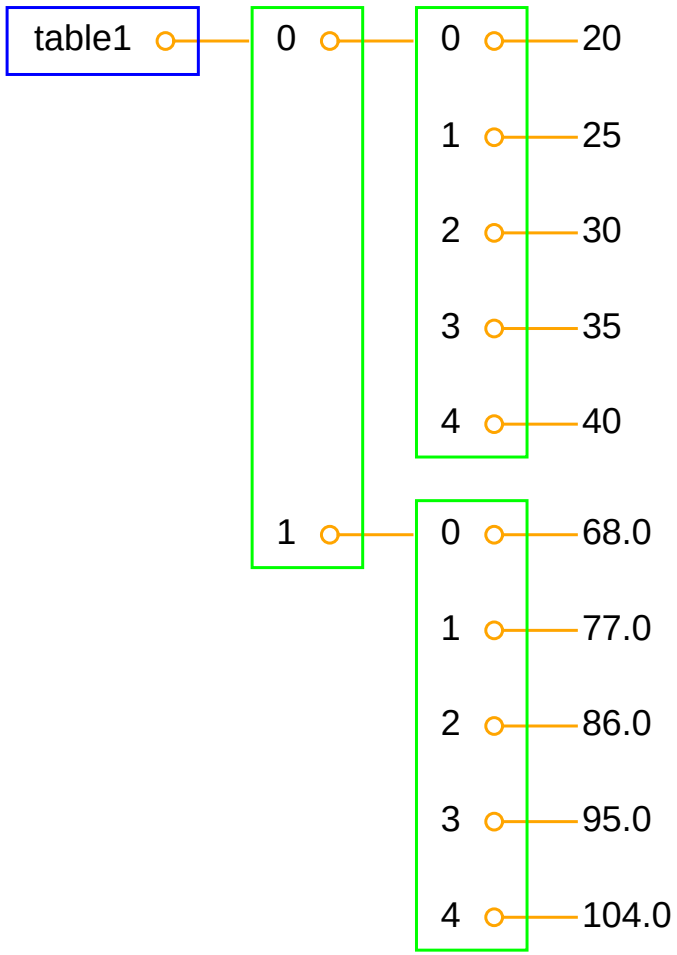
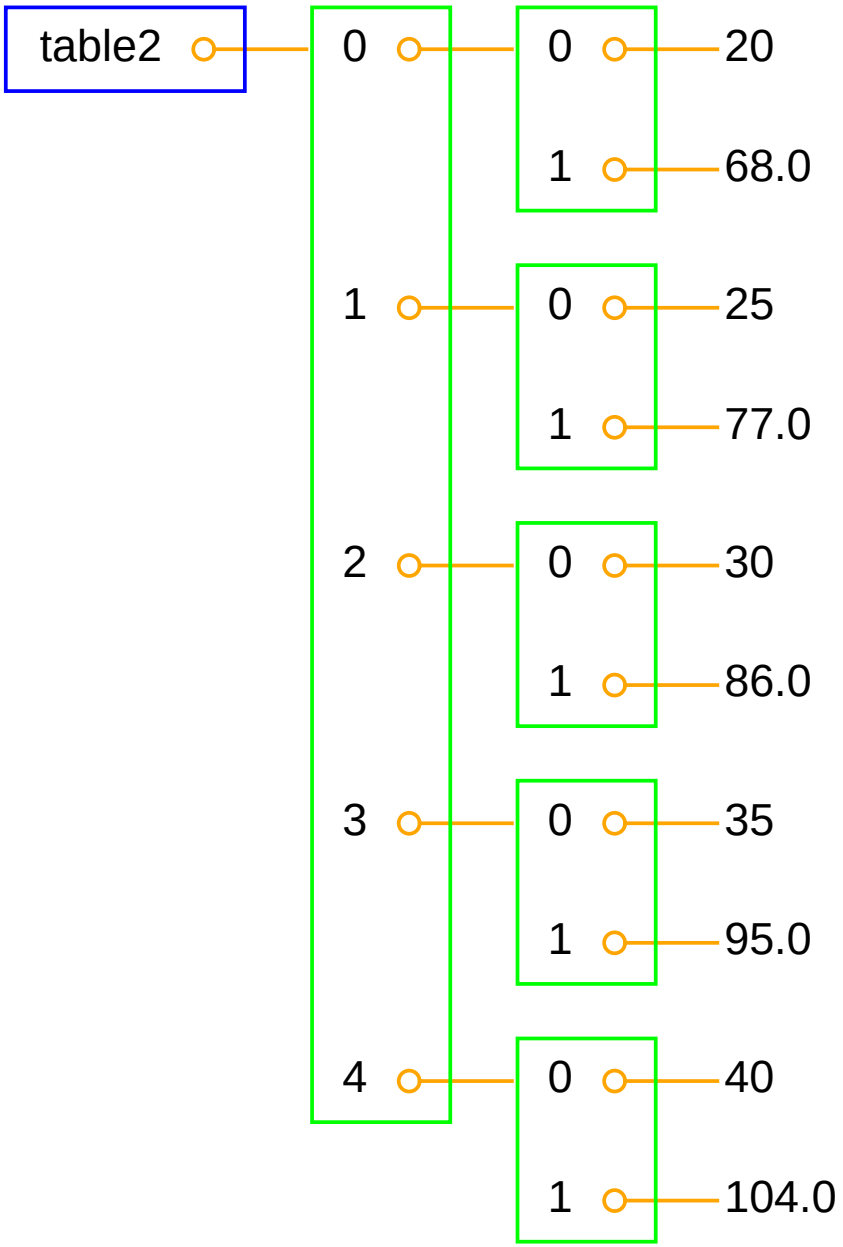


Illustration of table of rows



Shallow and Deep Copy

- Sharing = Saving = Out of Control
- Object copy

```
import copy
copy.copy(x) #returns a shallow copy of x
copy.deepcopy(x) #returns a deep copy
```

Exercise

- Problem set 3, Q. 1

What does this code snippet do?

```
for C, F in table2[Cdegrees.index(10):Cdegrees.index(35)]:
    print '%5.0f %5.1f' % (C, F)
```

- This is a for loop over a sublist of `table2`
- Sublist indices: `Cdegrees.index(10)`, `Cdegrees.index(35)`, i.e., the indices corresponding to elements 10 and 35, i.e., we want to run over rows in `table2` starting with the one where `C` is 10 and ending in the row before the row where `C` is 35
- Output:

```
   10  50.0
   15  59.0
   20  68.0
   25  77.0
   30  86.0
```

What does this code snippet do?

```
for C, F in table2[Cdegrees.index(10):Cdegrees.index(35)]:
    print '%5.0f %5.1f' % (C, F)
```

- This is a for loop over a sublist of `table2`
- Sublist indices: `Cdegrees.index(10)`, `Cdegrees.index(35)`, i.e., the indices corresponding to elements 10 and 35, i.e., we want to run over rows in `table2` starting with the one where `C` is 10 and ending in the row before the row where `C` is 35
- Output:

```
   10  50.0
   15  59.0
   20  68.0
   25  77.0
   30  86.0
```

What does this code snippet do?

```
for C, F in table2[Cdegrees.index(10):Cdegrees.index(35)]:
    print '%5.0f %5.1f' % (C, F)
```

- This is a for loop over a sublist of `table2`
- Sublist indices: `Cdegrees.index(10)`, `Cdegrees.index(35)`, i.e., the indices corresponding to elements 10 and 35, i.e., we want to run over rows in `table2` starting with the one where `C` is 10 and ending in the row before the row where `C` is 35
- Output:

```
10  50.0
15  59.0
20  68.0
25  77.0
30  86.0
```

More general nested lists

- We have seen one example on a nested list: a table with n rows, each row with a `[C, F]` list of two elements
- Traversal of this list (`table2`):

```
for C, F in table2:
    # work with C and F (columns in the current row)
```

- What if we have a more general list with m rows, n columns, and maybe not the same number of columns in each row?

Example: table with variable no of columns (1)

- We want to record the history of scores in a game
- Each player has played the game a certain number of times
- `scores[i][j]` is a nested list holding the score of game no. `j` for player no. `i`
- Some sample code:

```
scores = []
# score of player no. 0:
scores.append([12, 16, 11, 12])
# score of player no. 1:
scores.append([9])
# score of player no. 2:
scores.append([6, 9, 11, 14, 17, 15, 14, 20])
```

- Desired printing of `scores`:

```
12  16  11  12
 9
 6   9  11  14  17  15  14  20
```

- (Think of many players, each with many games)

Example: table with variable no of columns (2)

We use two loops: one over rows and one over columns

Loops over two list indices (integers):

```
for r in range(len(scores)):
    for c in range(len(scores[r])):
        score = scores[r][c]
        print '%4d' % score,
    print
```

Or: outer loop over rows, inner loop over columns:

```
for row in scores:
    for column in row:
        score = column   # better name...
        print '%4d' % score,
    print
```

Iteration of general nested lists

List with many indices: `somelist[i1][i2][i3]...`

Loops over list indices:

```
for i1 in range(len(somelist)):
    for i2 in range(len(somelist[i1])):
        for i3 in range(len(somelist[i1][i2])):
            for i4 in range(len(somelist[i1][i2][i3])):
                value = somelist[i1][i2][i3][i4]
                # work with value
```

Loops over sublists:

```
for sublist1 in somelist:
    for sublist2 in sublist1:
        for sublist3 in sublist2:
            for sublist4 in sublist3:
                value = sublist4
                # work with value
```

Exercise

- Problem set 3: Q4

Tuples: lists that cannot be changed

- Tuples are "constant lists":

```
>>> t = (2, 4, 6, 'temp.pdf')    # define a tuple
>>> t =  2, 4, 6, 'temp.pdf'     # can skip parenthesis
>>> t[1] = -1
...
TypeError: object does not support item assignment

>>> t.append(0)
...
AttributeError: 'tuple' object has no attribute 'append'

>>> del t[1]
...
TypeError: object doesn't support item deletion
```

- Tuples can do much of what lists can do:

```
>>> t = t + (-1.0, -2.0)           # add two tuples
>>> t
(2, 4, 6, 'temp.pdf', -1.0, -2.0)
>>> t[1]                           # indexing
4
>>> t[2:]                          # subtuple/slice
(6, 'temp.pdf', -1.0, -2.0)
>>> 6 in t                         # membership
True
```

Why tuples when lists have more functionality?

- Tuples are constant and thus protected against accidental changes
- Tuples are faster than lists
- Tuples are widely used in Python software
  (so you need to know about tuples!)
- Tuples (but not lists) can be used as keys is dictionaries
  (more about dictionaries later)

Summary of loops, lists and tuples

- Loops:

```
while condition:
    <block of statements>

for element in somelist:
    <block of statements>
```

- Lists and tuples:

```
mylist  = ['a string', 2.5, 6, 'another string']
mytuple = ('a string', 2.5, 6, 'another string')
mylist[1]  = -10
mylist.append('a third string')
mytuple[1] = -10  # illegal: cannot change a tuple
```

List functionality

`a = []`	initialize an empty list
`a = [1, 4.4, 'run.py']`	initialize a list
`a.append(elem)`	add `elem` object to the end
`a + [1,3]`	add two lists
`a[3]`	index a list element
`a[-1]`	get last list element
`a[1:3]`	slice: copy data to sublist (here: index 1, 2)
`del a[3]`	delete an element (index 3)
`a.remove(4.4)`	remove an element (with value `4.4`)
`a.index('run.py')`	find index corresponding to an element's value
`'run.py' in a`	test if a value is contained in the list
`a.count(v)`	count how many elements that have the value `v`
`len(a)`	number of elements in list `a`
`min(a)`	the smallest element in `a`
`max(a)`	the largest element in `a`
`sum(a)`	add all elements in `a`
`a.sort()`	sort list `a` (changes `a`)
`as = sorted(a)`	sort list `a` (return new list)
`a.reverse()`	reverse list `a` (changes `a`)
`b[3][0][2]`	nested list indexing
`isinstance(a, list)`	is `True` if `a` is a list

A summarizing example for Chapter 2; problem

- textttsrc/misc/Oxford_sun_hours.txt: data of the no of sun hours in Oxford, UK, for every month since Jan, 1929:

```
[
[43.8, 60.5, 190.2, ...],
[49.9, 54.3, 109.7, ...],
[63.7, 72.0, 142.3, ...],
...
]
```

- Compute the average number of sun hours for each month during the total data period (1929–2009)', r'Which month has the best weather according to the means found in the preceding task?
- For each decade, 1930-1939, 1949-1949, ..., 2000-2009, compute the average number of sun hours per day in January and December

A summarizing example for Chapter 2; the program (task 1)

```
data = [
[43.8, 60.5, 190.2, ...],
[49.9, 54.3, 109.7, ...],
[63.7, 72.0, 142.3, ...],
...
]
monthly_mean = [0]*12
for month in range(1, 13):
    m = month - 1   # corresponding list index (starts at 0)
    s = 0           # sum
    n = 2009 - 1929 + 1  # no of years
    for year in range(1929, 2010):
        y = year - 1929  # corresponding list index (starts at 0)
        s += data[y][m]
    monthly_mean[m] = s/n
month_names = 'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
# nice printout:
for name, value in zip(month_names, monthly_mean):
    print '%s: %.1f' % (name, value)
```

A summarizing example for Chapter 2; the program (task 2)

```
max_value = max(monthly_mean)
month = month_names[monthly_mean.index(max_value)]
print '%s has best weather with %.1f sun hours on average' %

max_value = -1E+20
for i in range(len(monthly_mean)):
    value = monthly_mean[i]
    if value > max_value:
        max_value = value
        max_i = i  # store index too
print '%s has best weather with %.1f sun hours on average' %
```

A summarizing example for Chapter 2; the program (task 3)

```
decade_mean = []
for decade_start in range(1930, 2010, 10):
    Jan_index = 0; Dec_index = 11  # indices
    s = 0
    for year in range(decade_start, decade_start+10):
        y = year - 1929  # list index
        print data[y-1][Dec_index] + data[y][Jan_index]
        s += data[y-1][Dec_index] + data[y][Jan_index]
    decade_mean.append(s/(20.*30))
for i in range(len(decade_mean)):
    print 'Decade %d-%d: %.1f' %           (1930+i*10, 1939+i*10, d
```

Exercise

- Problem set 3, Q. 5

MONTE CARLO SIMULATION

Random numbers are used to simulate uncertain events

- Some problems in science and technology are desrcribed by "exact" mathematics, leading to "precise" results
- Examples: throwing a ball, an oscillating system

- Some problems appear physically uncertain
- Examples: rolling a die, molecular motion
- Can we roll a die on a computer?
- Yes, by using *random numbers* to mimic the uncertainty of the experiment

- Random numbers make it possible to simulate physical systems with uncertainty, in input data or the process
- Random numbers are essential for programming games

Drawing random numbers

- Python has a `random` module for drawing random numbers
- `random.random()` draws random numbers in $[0, 1)$:

```
>>> import random
>>> random.random()
0.8155054685338104
>>> random.random()
0.4491332680902985 2
>>> random.random()
0.8832065311636745 4
```

- The sequence of random numbers is produced by a deterministic algorithm – the numbers just appear random

Distribution of random numbers

- `random.random()` generates random numbers that are *uniformly distributed* in the interval $[0, 1)$
- `random.uniform(a, b)` generates random numbers uniformly distributed in $[a, b)$
- "Uniformly distributed" means that if we generate a large set of numbers, no part of $[a, b)$ gets more numbers than others

Drawing integers

- Quite often we want to draw an integer from $[a, b]$ and not a real number
- Python's `random` module and `numpy.random` have functions for drawing uniformly distributed integers:

```
import random
r = random.randint(a, b)  # a, a+1, ..., b

import numpy as np
r = np.random.randint(a, b+1, N)          # b+1 is not included
r = np.random.random_integers(a, b, N)    # b is included
```

Example: throwing a die

- Any no of eyes, 1-6, is equally probable when you throw a die
- What is the chance of getting a 6?
- We make a program that simulates the process:

```
import random
N = 10000
eyes = [random.randint(1, 6) for i in range(N)]
six = 0  # counter for how many times we get 6 eyes
for outcome in eyes:
    if outcome == 6:
        six += 1
print 'Got six %d times out of %d' % (six, N)
```

- Probability: `six/N` (exact: 1/6)
- This is called Monte Carlo simulation

Fixing the seed fixes the random sequence

- Debugging programs with random numbers is difficult because the numbers produced vary each time we run the program
- For debugging it is important that a new run reproduces the sequence of random numbers in the last run
- This is possible by fixing the *seed* of the `random` module

```
random.seed(121)   # int argument
```

- The value of the seed determines the random sequence:

```
>>> import random
>>> random.seed(2)
>>> ['%.2f' % random.random() for i in range(7)]
['0.96', '0.95', '0.06', '0.08', '0.84', '0.74', '0.67']
>>> ['%.2f' % random.random() for i in range(7)]
['0.31', '0.61', '0.61', '0.58', '0.16', '0.43', '0.39']

>>> random.seed(2)    # repeat the random sequence
>>> ['%.2f' % random.random() for i in range(7)]
['0.96', '0.95', '0.06', '0.08', '0.84', '0.74', '0.67']
```

- By default, the seed is based on the current time

Drawing random elements from a list

- There are different methods for picking an element from a list at random, but the main method applies `choice(list)`:

```
>>> awards = ['car', 'computer', 'ball', 'pen']
>>> import random
>>> random.choice(awards)
'car'
```

- Alternatively, we can compute a random index:

```
>>> index = random.randint(0, len(awards)-1)
>>> awards[index]
'pen'
```

- We can also shuffle the list randomly, and then pick any element:

```
>>> random.shuffle(awards)
>>> awards[0]
'computer'
```

Probability via Monte Carlo simulation

- What is the probability that a certain event A happens?
- Simulate N events, count how many times M the event A happens, the probability of the event A is then M/N (as $N \to \infty$)
- Example: what is the probability of getting 6 on two or more dice if we throw 4 dice?

```
import random
N = 100000              # no of experiments
M = 0                   # no of successful events
for i in range(N):
    six = 0             # count the no of dice with a six
    r1 = random.randint(1, 6)
    if r1 == 6: six += 1
    ... same for dice 2, 3 and 4 ...
    # successful event?
    if six >= 2:
        M += 1
p = float(M)/N
print 'probability:', p
```

Game: guess a number

Game:

Let the computer pick a number at random. You guess at the number, and the computer tells if the number is too high or too low.

Program:

```
import random
number = random.randint(1, 100)
attempts = 0  # count no of attempts to guess the number
guess = 0
while guess != number:
    guess = input('Guess a number: ')
    attempts += 1
    if guess == number:
        print 'Correct! You used', attempts, 'attempts!'
        break
    elif guess < number:  print 'Go higher!'
    else:                 print 'Go lower!'
```

Exercise

- Write a program to approximate the value of using Monte Carlo simulation.

RECURSION

Exercise: Complete the Code

- Euclid's algorithm to calculate the greatest common divisor:

```
def gcd(a,b):
    if a == b:                  #if a equals b, gcd(a,b) is a
        ...
    elif a > b:                 #if a > b,  gcd(a,b) is gcd(a-b,b)

    else:                       #if a < b, gcd(a,b) is gcd(a,b-a)
```

Example

A prime number is a prime integer p, greater than 1, which is not evenly divisible by any positive integers other than p and 1. Examples are 17, 23, and 211. Any positive integer greater than 1 may be written as a product of primes. For example: 516 = 2 2 3 43. Write a recursive program which, given n, prints out the primes whose product is n.