Loops and lists

Foundation of programming (CK0030)

Francesco Corona

Alternative implementations

Loops and lists

Loops and lists

FfdP

Introduction to variables, objects, modules, and text formatting

Programming with WHILE- and FOR-loops, and lists

Functions and IF-ELSE tests

Data reading and writing

Error handling

Making modules

Arrays and array computing

Plotting curves and surfaces

Alternative implementations

Loops and lists

Usually, there are alternative ways to write code that solves a problem

- We explore alternative constructs and programs
- Store numbers in lists and print out tables
Loops and lists

Alternative implementations

WHILE loops as FOR loops
Range construction
FOR loops with list indexes
Modify list elements
List comprehension
Multiple lists
Nested lists
Tables as row/column lists
Printing objects
Extracting objects
Traversing nested lists
Some list operations
Tuples

WHILE loops as FOR loops

Alternative implementations

WHILE loops as FOR loops
Range construction
FOR loops with list indexes
Modify list elements
List comprehension
Multiple lists
Nested lists
Tables as row/column lists
Printing objects
Extracting objects
Traversing nested lists
Some list operations
Tuples

Definition

Any FOR-loop can be implemented as a WHILE-loop

Consider the general piece of code

```
for element in somelist:
    <process element>
```

It can be re-written

```
index = 0
while index < len(somelist):
    element = somelist[index]
    <process element>
    index += 1
```

Example

Printout of the Celsius-Fahrenheit table of temperatures

```
Cdegrees = [-20,-15,-10,-5,0,5,10,15,20,25,30,35,40]
print 'C  F'
index = 0
while index < len(Cdegrees):
    C = Cdegrees[index]
    F = (9.0/5)*C+32
    print '%5d %5.1f' % (C,F)
    index += 1
```

```
C = -20
dC = 5
while C <= 40:
    F = (9.0/5)*C+32
    print C, F
    C = C + dC
```
It is often tedious to manually type the many elements in `degrees`

```
C_value = -50
C_max = 200
Cdegrees = []
```

```
while C_value <= C_max:
    Cdegrees.append(C_value)
    C_value += 2.5
```

The `range construction` is a particularly useful tool for the task.

### Range construction (cont.)

#### Definition

**range(n)**  
generates a list of sequential integers in \([0, n - 1]\)  
- (Integer `n` is not included)  
\(\sim 0, 1, 2, \ldots, n-1\)

**range(start, stop, step)** generates a list of integers in a sequence  
\(\sim \text{start}, \text{start} + (1*\text{step}), \text{start} + (2*\text{step})\) up to stop  
- (stop is not included)

`range(start, stop) is the same as range(start, stop, 1)`  
\(\sim \text{start}, \text{start} + (1*1), \text{start} + (2*1)\) up to stop  
- (That is, step = 1)

### Range construction (cont.)

**Things to remember**

In Python 2.x, function `range(n)` returns a list object

In Python 3.x, function `range(n)` returns a range object  
- A range object can be converted to a list object  
\(\sim \text{list(range(n))}\)

This exists in Python 2.x as function `xrange(n)`
Loops and lists

Range construction (cont.)

Example

Consider the following examples

\[ \text{range}(2, 8, 3) \]
- The output
  \[ \sim 2 \]
  \[ \sim 2 + (1\cdot3) = 5 \text{ (but not } 8 = 2 + (2\cdot3)) \]

\[ \text{range}(1, 11, 2) \]
- The output
  \[ \sim 1 \]
  \[ \sim 1 + 1 = (1\cdot2) \]
  \[ \sim 5 = 1 + (2\cdot2) \]
  \[ \sim 7 = 1 + (3\cdot2) \]
  \[ \sim 9 = 1 + (4\cdot2) \]

Loops and lists

Range construction (cont.)

Example

We use \texttt{range} to create a list \texttt{Cdegrees} with values \([-20,-15,\ldots,35,40]\)
- Two ways (with and without a loop)

\[
\texttt{Cdegrees} = [] \quad \# \text{ Create empty list to be filled}
\]

\[
\texttt{Cdegrees} = \texttt{range}(-20, 45, 5) \quad \# \text{ Pick element } C \text{ from a list of sequential integers}
\]

\[
\texttt{Cdegrees} = \texttt{range}(-20, 45, 5) \quad \# \text{ Create list } C \text{ inside the FOR loop}
\]

\[
\texttt{Cdegrees} = \texttt{range}(-20, 45, 5) \quad \# \text{ 1st element: } -20
\]
\[
\texttt{Cdegrees} = \texttt{range}(-20, 45, 5) \quad \# \text{ 2nd element: } -20 + (1\cdot5) = -15
\]
\[
\texttt{Cdegrees} = \texttt{range}(-20, 45, 5) \quad \# \text{ 3rd element: } -20 + (2\cdot5) = -10
\]
\[
\texttt{Cdegrees} = \texttt{range}(-20, 45, 5) \quad \# \ldots
\]

To include integer 40, the upper limit must be greater than 40
- This is important

Loops and lists

Range construction (cont.)

Example

Suppose that now we want to create a slightly different \texttt{Cdegrees} list
- \([-10, -7.5, -5, \ldots, 35, 37.5, 40]\)
- The spacing between entries is 2.5
- The entries are real numbers

We cannot use \texttt{range} directly, we must adapt its use
- \texttt{range}(-10, 45, 2.5) would give an error
- \texttt{range} can only create integers
- We have decimal degrees
Loops and lists

Alternative implementations
WHILE loops
Range construction
FOR loops with list indexes
Modify list elements
List comprehension
Multiple lists
Nested lists
Tables as row/column lists
Printing objects
Extracting elements
Traversing nested lists
Some list operations
Tuples

FOR loops with list indexes (cont.)

Consider an alternative to iterating over (the elements of) a list directly

```python
for element in somelist:
    ... # Some operation on element
    ... # <Process element>
```

We can iterate over list indices and then index the list inside the loop

```python
for i in range(len(somelist)):
    ... # Some operation on element
    ... # <Process element>
```

len(somelist) returns the length of somelist

- Indices start at 0, the largest valid index is len(somelist)-1
- range(len(somelist)) is [0, 1, ..., len(somelist)-1]

Loops and lists

Alternative implementations
WHILE loops
Range construction
FOR loops with list indexes
Modify list elements
List comprehension
Multiple lists
Nested lists
Tables as row/column lists
Printing objects
Extracting elements
Traversing nested lists
Some list operations
Tuples

FOR loops with list indexes

Range construction (cont.)

We must introduce an integer counter `i` generate by function `range`

- We generate `C` values by `C = -10 + 2.5 \cdot i`, `i = 0, 1, 2, \ldots, 20`

```python
Cdegrees = []
for i in range(0, 21):
    # Generate a range of integers
    C = -10 + i \cdot 2.5
    # Element i is used here
    Cdegrees.append(C)
```

Iterating over loop indices is often a useful programming practice

- An example is when we need to process two lists
- (At the same time)
### Example

Suppose that we want to create two lists, C\text{degrees} and F\text{degrees}.

Then, suppose that we want to use the two lists to write a table:

- The table must have C\text{degrees} and F\text{degrees} as columns.

```python
n = 21
C_{\text{min}} = -10; C_{\text{max}} = 40
\text{Inc} = (C_{\text{max}} - C_{\text{min}})/\text{float}(n-1)

C\text{degrees} = [0]*n  # Build the C list
F\text{degrees} = [0]*n  # Build the F list

for i in range(len(C\text{degrees})):  # Print the joint table
    print '%5.1f %5.1f %5.1f %5.1f' % (C\text{degrees}[i], F\text{degrees}[i], C_i, F_i)
```

In the example, we started with empty lists then appended new elements. We can start with lists of correct size, containing, say, zeros:

- Then, we index the lists to fill in actual values.
Modify list elements

Alternative implementations

Consider some list of temperature values accessible with name `Cdegrees`

Suppose that we want to change the value of each of its elements

- We want to add 5 (degrees)

```python
m = 2; C_min = -10; C_max = 40
dc = (C_max - C_min) / float(m - 1)
Cdegrees = []
for i in range(len(Cdegrees)):  # Adjust the i-th element to be equal to itself plus five
    Cdegrees[i] += 5
```

Things that do NOT work

- Variable `c` can only be used to read list elements
  ~ It does not change them
  ~ Only `c` is changed

Things that DO work

```python
for c in Cdegrees:
    c += 5
```

Remark

To change a list element, `Cdegrees[i]`, an assignment must be used

```python
Cdegrees[i] = ...  # Change the i-th list element
```
Loops and lists

**List comprehension**

- Run thru a list and for each element create a new element in another list
  - This is a frequently encountered task
  - \((F_{\text{degrees}}[i] \text{ from } C_{\text{degrees}}[i])\)

Python has a special compact syntax for this

- **List comprehension**

List comprehension (cont.)

The general syntax for list comprehension

\[
\text{newlist} = [E(e) \text{ for } e \text{ in list}]
\]

- **E(e)** is some expression involving element **e** of list **list**

Consider the following code, the tasks should be familiar

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
</table>
| C_{\text{degrees}} = [-5+i*0.5 for i in range(n)] | # List comprehension
| F_{\text{degrees}} = [(9.0/5)*C+32 for C in C_{\text{degrees}}] | # List comprehension
| C_{\text{plus}_5} = [C+5 for C in C_{\text{degrees}}] | # Build list C_{\text{plus}_5}

How does the computation evolve in each case?

What are the elements of the lists?
Travessing multiple lists

Suppose that we want to use lists Cdegrees and Fdegrees to make a table

- We need to traverse both arrays

A for element in list construction is not suitable here

- It extracts elements from one list only

A solution is to use a FOR-loop over indices

- So that we can index both lists
- (We silently used this already)

Travessing multiple lists (cont.)

It often happens that two or more lists need be traversed simultaneously

Python offers an alternative to the loop over indices

- A special syntax
  - The zip function

Example

Consider this piece of code for printing a table of temperature values

```
# List comprehension
Cdegrees = [-5+i*0.5 for i in range(n)]
Fdegrees = [(9.0/5)*C+32 for C in Cdegrees]

for i in range(len(Cdegrees)):
    print '%5d %5.1f' % (Cdegrees[i], Fdegrees[i])
```

Travessing multiple lists (cont.)

Definition

**Zip**

Function zip turns n lists (list1, list2, ...) into a single list of n-tuples

```
for e1, e2, ... in zip(list1, list2, ...):
    # Element e1 from list1
    # Element e2 from list2
    ...
```

For each n-tuple (e1, e2, ...),

- The first element (e1) is from the first list (list1)
- The second element (e2) is from second list (list2)
- ...
- The n-th element (en) is from second list (listn)

The loop stops when the end of the shortest list is reached
Traversing multiple lists (cont.)

Consider the following code using list comprehension and the `zip` function:

```python
n = 21
Cdegrees = [-5+i*0.5 for i in range(n)]
# List comprehension
Fdegrees = [(9.0/5)*C+32 for C in Cdegrees]
# List comprehension
for C, F in zip(Cdegrees, Fdegrees):
    # Print temperatures
    print('%5.1f %5.1f' % (C, F))
```

The result of the execution of the first part of the code:

```
>>> Cdegrees
[-5.0, -4.5, -4.0, -3.5, -3.0, -2.5, -2.0, -1.5, -1.0, -0.5, 0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0]
```

```
>>> Fdegrees
[23.0, 23.9, 24.8, 25.7, 26.6, 27.5, 28.4, 29.3, 30.2, 31.1, 32.0, 32.9, 33.8, 34.7, 35.6, 36.5, 37.4, 38.3, 39.2, 40.1, 41.0, 41.9]
```

Consider the continuation code using the `zip` function and list comprehension:

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

The result of the execution of the second part of the code:

```
>>> table
[[-5.0, 23.0], [-4.5, 23.9], [-4.0, 24.8], [-3.5, 25.7], [-3.0, 26.6], [-2.5, 27.5], [-2.0, 28.4], [-1.5, 29.3], [-1.0, 30.2], [-0.5, 31.1], [0.0, 32.0], [0.5, 32.9], [1.0, 33.8], [1.5, 34.7], [2.0, 35.6], [2.5, 36.5], [3.0, 37.4], [3.5, 38.3], [4.0, 39.2], [4.5, 40.1], [5.0, 41.0]]
```
### Nested lists

**Loops and lists**

#### Nested lists

- Loops and lists
- Alternative implementations
- WHILE loops as FOR loops
- Range construction
- FOR loops with list indexes
- Modify list elements
- List comprehension
- Multiple lists
- Tables as row/column lists
- Printing objects
- Extracting sublists
- Traversing nested lists
- Some list operations
- Tuples

---

**A table as a row or column list**

- **Nested lists**

  We use some examples to motivate the need for nested lists
  - We shall also illustrate some basic operations

  - **Nested lists are list objects, the list elements are list objects**

  In our table of temperatures, we used a separate list for each table column

  - With \( n \) columns, we need \( n \) list objects to handle table data
    
    ```python
    n=21
    Cdegrees = [-5+i*0.5 for i in range(n)]
    Fdegrees = [(9.0/5)*C+32 for C in Cdegrees]
    Kdegrees = [C+273.15 for C in Cdegrees]
    table = []
    table = [[C,F,K] for C,F,K in zip(Cdegrees, Fdegrees, Kdegrees)]
    ```
Loops and lists
FC
CK0030
2018.1

Alternative implementations
WHILE loops vs.
FOR loops
Range construction
FOR loops with list
indexes
Modify list elements
List comprehension
Multiple lists
Nested lists
Tables as
row/column lists
Printing objects
Extracting sublists
Travessing nested
lists
Some list operations
Tuples

A table as a row or column list

We think of a table as a single entity, not a collection of n columns
• It is natural to use one argument for the whole table

In Python this can be achieved by using a nested list
• Each entry in the list is a list itself

Example

Cdegrees = range(-20, 41, 5)  # -20, -15, ..., 35, 40
Fdegrees = [(9.0/5)*C + 32 for C in Cdegrees]
table = [Cdegrees, Fdegrees]

~ The table is a list of two columns
~ Each column is a list of numbers

A table as a row or column list (cont.)

A table object is understood as a list of lists

We can see it as two different cases
• Either it is a list of the row elements of the table
• Or, it is a list of the column elements of the table

A table as a row or column list (cont.)

A table as a row or column list (cont.)
### A table as a row or column list (cont.)

Consider tabular data with rows and columns:
- The underlying data are a nested list
- The first index counts the rows
- The second index counts the columns

This is the convention for indexing elements.

#### Example

We can construct `table` as a list of `[C, F]` pairs:
- The first index will then run over rows `[C, F]`

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

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

This construction is based on looping through pairs `C` and `F`:
- At each pass, we create a list element `[C, F]`
- Then, we append it as last element to `table`
Loops and lists
FC
CK0030
2018.1
Alternative implementations
WHILE loops vs.
FOR loops
Range construction
FOR loops with list
indexes
Modify list elements
List comprehension
Multiple lists
Nested lists
Tables as
row/column lists
Printing objects
Extracting tables
Traversing nested lists
Some list operations
Tuples

A table as a row or column list (cont.)
FC
CK0030
2018.1
Alternative implementations
WHILE loops vs.
FOR loops
Range construction
FOR loops with list
indexes
Modify list elements
List comprehension
Multiple lists
Nested lists
Tables as
row/column lists
Printing objects
Extracting tables
Traversing nested lists
Some list operations
Tuples

A list of columns and a list of pairs

```
| Cdegrees = range(-20, 41, 5) |
| Fdegrees = [(9.0/5)*C + 32 for C in Cdegrees] |
| table = [] |
| for C, F in zip(Cdegrees, Fdegrees): |
| table.append([C, F]) |
```

```
>>> Cdegrees |
[-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40] |
>>> Fdegrees |
[-4.0, 5.0, 14.0, 23.0, 32.0, 41.0, 50.0, 59.0, 68.0, 77.0, 86.0, 95.0] |
>>> table |
[[-20, -4.0], [-15, 5.0], [-10, 14.0], [-5, 23.0], [0, 32.0], [5, 41.0], [10, 50.0], [15, 59.0], [20, 68.0], [25, 77.0], [30, 86.0], [35, 95.0], [40, 104.0]] |
```

This construction is based on looping through pairs C and F
• At each pass, we create a list element [C, F]
• (The process of appending it not explicit)

The first index looks up an element in the outer list
• This element can be indexed with the second index

```
  • With table[5][0], we access the C value
  • With table[5][1], we access the F value
```
Printing objects

To immediately view the nested list `table`, we may write `print table`
- Any object `obj` can be printed to screen by `print obj`
The output is usually one line, which may be very long with packed lists

Example

A long list, like the `table` variable, needs a long line when printed

```
[[-20, -4.0], [-15, 5.0], [-10, 14.0], ..., ..., [40, 104.0]]
```

Splitting the output over shorter lines makes the layout more readable

Printing objects (cont.)

The book offers a modified `pprint` module, named `scitools.pprint2`
- Format control over printing of `float` objects in `list` objects
- `scitools.pprint2.float_format`, as `printf` format string

Example

How the output format of real numbers can be changed

```
>>> import pprint, scitools.pprint2
>>> someList = [15.8, [0.2, 1.7]]
>>> pprint.pprint(someList)
[15.800000000000001, [0.20000000000000001, 1.7]]
```

```python
>>> scitools.pprint2.pprint(someList)
[15.8, [0.2, 1.7]]
```

```python
>>> scitools.pprint2.float_format = '%.2e'
```

```python
>>> scitools.pprint2.pprint(someList)
[1.58e+01, [2.0e-01, 1.7e+00]]
```
Loops and lists

Printing objects (cont.)

Definition

\texttt{pformat} and \texttt{scitools.pprint2} modules have function \texttt{pformat}

\begin{itemize}
\item It returns a formatted string, rather than printing a string
\item It works as \texttt{pprint}
\end{itemize}

Example

Loop over each row, extract the two elements $C$ and $F$ in each row

\begin{itemize}
\item Print these in fixed width fields
\item Use the \texttt{printf} syntax
\end{itemize}

Extracting sublists

Python has a syntax for extracting/accessing parts of a list structure

\begin{itemize}
\item Sublists or slices
\end{itemize}
**Extracting sublists (cont.)**

A\([i:]\) refers to the sublist of A starting with index i in A till the end of A

```python
>>> A = [2, 3.5, 8, 10]
# 0 1 2 3
>>> A[2:]
[8, 10]
```

A\([i:]\) refers to the sublist of A starting with index of 0 in A till index i-1

```python
>>> A = [2, 3.5, 8, 10]
# 0 1 2 3
>>> A[:3]
[2, 3.5, 8]
```

- The last index that is considered is \(i-1\)
- (This is important to remember)

**Extracting sublists (cont.)**

A\([1:j]\) refers to the sublist of A starting with index i in A till index j-1

```python
>>> A = [2, 3.5, 8, 10]
# 0 1 2 3
>>> A[1:3]
[3.5, 8]
```

- The last index that is considered is \(j-1\)
- (This is important to remember)

**Extracting sublists (cont.)**

A\([i:-1]\) extracts all elements except the first and the last

- (Index \(-1\) refers to the last element)

```python
>>> A = [2, 3.5, 8, 10]
# 0 1 2 3
>>> A[1:-1]
[3.5, 8]
```

**Extracting sublists (cont.)**

A\([\_\_\_\_\_\_\_\_\_\_\_\_]\) refers to the whole list

```python
>>> A = [2, 3.5, 8, 10]
```

With nested lists, it is possible to use slices in the first index

```python
>>> table[4:]  
[[[-20, -4.0], # table [0]  
[-15, 5.0], # table [1]  
[-10, 14.0], # table [2]  
[-5, 23.0], # table [3]  
[0, 32.0], # table [4]  
[5, 41.0], # table [5]  
[10, 50.0], # table [6]  
[15, 59.0], # table [7]  
[20, 68.0], # table [8]  
[25, 77.0], # table [9]  
[30, 86.0], # table [10]  
[35, 95.0], # table [11]  
[40, 104.0]] # table [12]
```
Extracting sublists (cont.)

```python
>>> table = [[-20, -4.0], [-15, 5.0], [-10, 14.0], [-5, 23.0], [0, 32.0], [5, 41.0], [10, 50.0], [15, 59.0], [20, 68.0], [25, 77.0], [30, 86.0], [35, 95.0], [40, 104.0]]
```

We can also slice the second index, or both indices

```python
>>> table[4:7][0:2]
[[0, 32.0], [5, 41.0]]
```

Slice [0:2] acts on it, picks its first two elements

```python
>>> table = table[4:7][0:2]
```

```
0:32.0, 5:41.0, 10:50.0
```

Remark

Suppose that you have pre-defined/available some list

- Suppose that you extract some sublist from it
- Suppose that you modify such sublist

Whatever the modification on the sublist, the original list remains unaltered

- The *vice versa* is also true

Extracting sublists (cont.)

Sublists are always copies of the original list

- This is important

Example

```python
>>> list_1 = [1, 4, 3]
# Defini list_1
>>> list_2 = list_1[:-1]
# It is a sublist of list_1
# Elements 0 to -2
>>> list_2
[1, 4]
# First element of list_1
>>> list_1[0] = 100
# List_1 is modified
>>> list_1
[100, 4, 3]
>>> list_2
[1, 4]
# List_2 is not modified
```

Extracting sublists (cont.)

Remark

\[ B \equiv A \] is True if all elements in \( B \) equal corresponding elements in \( A \)

The test \( B \equiv A \) is True if \( A \) and \( B \) are names for the same list
Extracting sublists (cont.)

Example

Consider the following piece of code

```python
>>> A = [2, 3.5, 8, 10]
>>> B = A[:]
>>> C = A

Setting B = A[:] makes B refer to a copy of the list referred to by A
Setting C = A makes C refer to the same list object as A
```

Extracting sublists (cont.)

Example

Write the part of the table list of \([C, F]\) rows where the degrees Celsius are between 10 and 35 (not including 35)

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

Travessing nested lists

Traversing the nested list `table` could be done by a loop

```python
for C, F in table:
    process C and F
```

Natural, when we know that `table` is a list of \([C, F]\) lists

More general nested lists must be handled differently

- Unknown how many elements there are in each list
- (Lists are the element of the main list)
Travessing nested lists (cont.)

Example

Consider a nested list \texttt{scores} recording the scores of players in some game
- \texttt{scores[i]} holds the list of scores obtained by player number \texttt{i}

Different players have played the game a different number of times
- The length of \texttt{scores[i]} depends on \texttt{i}, the player

```py
scores = []
# Hypothetical scores of player no. 0:
# Length 4
scores.append([12, 16, 11, 12])
# Hypothetical scores of player no. 1:
# Length 1
scores.append([9])
# Hypothetical scores of player no. 2:
# Length 8
scores.append([6, 9, 11, 14, 17, 15, 14, 20])
```

The list has three elements, each element corresponds to a player.

Travessing nested lists (cont.)

Remark

Consider \( n \) players, some may have played a large number of times

This makes \texttt{scores} a big nested list, potentially

```py
# Hypothetical scores of player no. 0:
scores.append([12, 16, 11, 12])
# Length 4
# Hypothetical scores of player no. 1:
scores.append([9])
# Length 1
# Hypothetical scores of player no. 2:
scores.append([6, 9, 11, 14, 17, 15, 14, 20])
```

Consider element number \( g \) in the list \texttt{scores[p], scores[p][g]}
- It corresponds to the score in game \( g \) played by player \( p \)

The length of the individual lists \texttt{scores[p]} varies
- It equals 4, 1, and 8 for \( p \) equal 0, 1, and 2, respectively

How to traverse the list and put it in table format

With well formatted columns?

The desired properties of the table formatting
- Each row must correspond to a player
- Columns must correspond to scores
Traversing nested lists (cont.)

We may use two nested loops
- One loop for the elements in `scores`
- One loop for the elements in the sublists of `scores`

There are two basic ways of traversing a nested list
- We use integer indices for each index
- We use variables for the list elements

We used the trailing comma after `print string`

```
scores = []
scores.append([12, 16, 11, 12])
scores.append([9])
scores.append([6, 9, 11, 14, 17, 15, 14, 20])
for p in range(len(scores)):
    for g in range(len(scores[p])):
        score = scores[p][g]
        print '%4d' % score,
p
```

The `print` after the loop over `p` adds a new (empty) line after each row.
Travessing nested lists (cont.)

Definition

Consider the general case of nested lists with many indices
~ somelist [i1][i2][i3] ...

Suppose that we are interested in visiting each element in the list
We can use as many nested FOR-loops as there are indices

Travessing nested lists (cont.)

As a practical example consider a nested list with four indices

```python
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]
                # perform some operation with this current value
```

This is what iterating over integer indices looks like

Travessing nested lists (cont.)

The corresponding version by iterating over sublists

```python
for sublist1 in somelist:
    for sublist2 in sublist1:
        for sublist3 in sublist2:
            for sublist4 in sublist3:
                value = sublist4
                # perform some operation with this current value
```

Some list operations

Tuples
Some list operations

<table>
<thead>
<tr>
<th>Construct</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a = []</code></td>
<td>Initialize an empty string</td>
</tr>
<tr>
<td><code>a = [1, 4.4, 'run.py']</code></td>
<td>Initialize a list</td>
</tr>
<tr>
<td><code>a.append(elem)</code></td>
<td>Add element</td>
</tr>
<tr>
<td><code>a = [1.3]</code></td>
<td>Add two lists</td>
</tr>
<tr>
<td><code>a.insert(i, e)</code></td>
<td>Insert element <code>e</code> before index <code>i</code></td>
</tr>
<tr>
<td><code>a[3]</code></td>
<td>Index a list element</td>
</tr>
<tr>
<td><code>a[-i]</code></td>
<td>Get last lists element</td>
</tr>
<tr>
<td><code>a[1:3]</code></td>
<td>Slide: Copy data to sublist</td>
</tr>
<tr>
<td><code>del a[3]</code></td>
<td>Delete an element</td>
</tr>
<tr>
<td><code>a.remove(e)</code></td>
<td>Remove an element with value <code>e</code></td>
</tr>
</tbody>
</table>

Some list operations (cont.)

<table>
<thead>
<tr>
<th>Construct</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a.index('run.py')</code></td>
<td>Index corresponding to element’s value</td>
</tr>
<tr>
<td><code>'run.py' in a</code></td>
<td>Test if a value is in the list</td>
</tr>
<tr>
<td><code>a.count(v)</code></td>
<td>Count elements with value <code>v</code></td>
</tr>
<tr>
<td><code>len(a)</code></td>
<td>Number of elements in list <code>a</code></td>
</tr>
<tr>
<td><code>min(a)</code></td>
<td>The smallest element in list <code>a</code></td>
</tr>
<tr>
<td><code>max(a)</code></td>
<td>The largest element in list <code>a</code></td>
</tr>
<tr>
<td><code>sum(a)</code></td>
<td>Add all elements in <code>a</code></td>
</tr>
<tr>
<td><code>reversed(a)</code></td>
<td>Return returned version of <code>a</code></td>
</tr>
<tr>
<td><code>b[3][0][2]</code></td>
<td>Nested list indexing</td>
</tr>
</tbody>
</table>

Tuples

**Tuples**

Lists and loops

Tuples are similar to lists, but **tuple objects** cannot be changed

- A tuple object can be viewed as a constant list object

Lists use square brackets, tuples employ standard parentheses
A comma-separated sequence of objects is a **tuple object**

- Parentheses are not necessary, though common

### Tuples (cont.)

Many of the usual functionalities for lists are also available for tuples

```
>>> t = (2, 4, 6, 'temp.pdf')  # Define a tuple
>>> t  # Name t
(2, 4, 6, 'temp.pdf')
```

```
>>> t = t + (-1.0, -2.0)  # Add two tuples
>>> t  # Name 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
```

We can use **FOR-loop** to loop over a tuple

```
for element in 'myfile.txt', 'urfile.txt', 'herfile.txt':
    print element,
```

Note the trailing comma (,) in the `print` statement

```
myfile.txt yourfile.txt herfile.txt
```

The comma suppresses the final newline that `print` command would add

- The output of `print` is a **string object**

### Tuples (cont.)

Operations for lists that change the list do not work for tuples

```
>>> t[1] = -1
  ...  # Indexing
TypeError: object does not support item assignment
```

```
>>> t.append(0)
  ...  # Add an item
AttributeError: 'tuple' object has no attribute 'append'
```

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

Some methods for lists (like `index`) are not available for tuples
So why do we need tuples at all when lists can do more than tuples?

~ Tuples protect against accidental changes of their contents
~ Code based on tuples is faster than code based on lists
~ Tuples are often used in Python software that you will use
  • (You need to know this data type!)

There is also a fourth argument, the data-type called dictionaries

• Tuples can be used as keys in dictionaries
• Lists cannot