Loops and lists

Foundation of programming (CK0030)

Francesco Corona

FdP

- ☺ Intro 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

FdP (cont.)

We discuss how repetitive tasks in a program are automated by **loops**

- We introduce **list objects** for storing and processing collections of data with a specific order
- Loops and lists, together with functions and IF-tests (soon) lay the fundamental programming foundation

WHILE loops
Loops and lists

WHILE loops

Example

We want to print out a conversion table with degree Celsius in the first column of the table and corresponding Fahrenheits in the second one

```
1  -20   -4.0
2  -15    5.0
3  -10   14.0
4   -5   23.0
5    0   32.0
6    5   41.0
7   10   50.0
8   15   59.0
9   20   68.0
10  25   77.0
11  30   86.0
12  35   95.0
13  40  104.0
```

The formula for converting C degrees Celsius to F degrees Fahrenheit is

$$F = \frac{9}{5}C + 32$$

We already know how to evaluate the formula for one single value of C

- We could repeat the statements as many times as required

WHILE loops (cont.)

- We repeatedly write the whole command (`c2f_table_repeat.py`)

```
1  C = -20; F = 9.0/5*C + 32; print C, F
2  C = -15; F = 9.0/5*C + 32; print C, F
3  C = -10; F = 9.0/5*C + 32; print C, F
4  C = -5;  F = 9.0/5*C + 32; print C, F
5  C = 0;   F = 9.0/5*C + 32; print C, F
6  C = 5;   F = 9.0/5*C + 32; print C, F
7  C = 10;  F = 9.0/5*C + 32; print C, F
8  C = 15;  F = 9.0/5*C + 32; print C, F
9  C = 20;  F = 9.0/5*C + 32; print C, F
10 C = 25;  F = 9.0/5*C + 32; print C, F
11 C = 30;  F = 9.0/5*C + 32; print C, F
12 C = 35;  F = 9.0/5*C + 32; print C, F
13 C = 40;  F = 9.0/5*C + 32; print C, F
```

- We use three statements per line in the code, for compact layout

WHILE loops (cont.)

Running this program shows how the output looks like on screen

```
1  -20  -4.0
2  -15  5.0
3  -10  14.0
4  -5  23.0
5   0  32.0
6   5  41.0
7   10  50.0
8   15  59.0
9   20  68.0
10  25  77.0
11  30  86.0
12  35  95.0
13  40  104.0
```

Remark

This output suffers from a rather primitive text formatting

- This can quickly be changed by replacing `print C, F` by a `print statement` based on `printf formatting`

WHILE loops (cont.)

The problem with this code is that identical statements are repeated

- It is boring and dumb to write those repeated statements, especially if we have more C and F values in the table
- One of the ideas behind a computer is to automate repetitions

All computer languages have constructs to efficiently express repetition

WHILE loops (cont.)

Such constructs are called **loops** and come in two variants in Python

- **WHILE-loops**
- **FOR-loops**

Remark

Most programs employ loops: It is fundamental to learn this concept

WHILE loops
WHILE loops

WHILE loops

A `WHILE-loop` is used to repeat a set of statements

- as long as a some condition is verified (it is true)

To discuss this loop, we use the temperature table

Example

The task is to generate the rows of the table of C and F values

```
-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
```

C values start at -20 and are incremented by 5, as long as $C \leq 40$

WHILE loops (cont.)

```
-20   -4.0
-15    5.0
-10   14.0

...   ....

 35   95.0
 40  104.0
```

For each C value, we compute the corresponding F value

$$F = \frac{9}{5}C + 32$$

- Then, we write out the two temperatures

We also add a line of dashes (-), above and below table

WHILE loops (cont.)

```
-20   -4.0
-15    5.0
-10   14.0

...   ....

 35   95.0
 40  104.0
```

The list of tasks to be done can be summarised as

1. Print line with dashes
2. Let $C = -20$
3. WHILE $C \leq 40$:
   - $F = \frac{9}{5}C + 32$
   - Print C and F
   - Increment C by 5
4. Print line with dashes

This is the **algorithm** of our programming task

WHILE loops (cont.)

Converting a detailed algorithm into a functioning code is often easy

```
1 print '------------------' # table heading
2 C = -20                   # start value for C
3 dC = 5                    # increment of C in loop
4 while C <= 40:            # loop heading with condition
5  F = (9.0/5)*C + 32       # 1st statement inside loop
6  print C, F               # 2nd statement inside loop
7  C = C + dC               # 3rd statement inside loop
8 print '------------------' # end of table line (after loop)
```

1. Print line with dashes
2. Let $C = -20$ (and $\Delta C = 5$)
3. WHILE $C \leq 40$:
   - $F = \frac{9}{5}C + 32$
   - Print C and F
   - Increment C by ($\Delta C =$) 5
4. Print line with dashes

WHILE loops (cont.)

The **block of statements** is executed at each pass of the `WHILE-loop`

- It must be indented

```
1  print '------------------' # table heading
2  C = -20                   # start value for C
3  dC = 5                    # increment of C in loop
4  while C <= 40:            # loop heading with condition
5
6   F = (9.0/5)*C + 32       # 1st statement inside loop
7   print C, F               # 2nd statement inside loop
8   C = C + dC               # 3rd statement inside loop
9
10 print '------------------' # end of table line (after loop)
```

The block is three lines, and all must have the same indentation

- Our choice of indentation is one (usually, four) spaces

Remark

The first statement whose indentation coincides with that of the `while` line marks the end of the loop and it is executed when the loop is done

- Here, this is the final `print statement`

WHILE loops (cont.)

What if in the code we also indent the last line one space?

```
1 print '------------------'  # table heading
2 C = -20                    # start value for C
3 dC = 5                     # increment of C in loop
4 while C <= 40:             # loop heading with condition
5  F = (9.0/5)*C + 32        # 1st statement inside loop
6  print C, F                # 2nd statement inside loop
7  C = C + dC                # 3rd statement inside loop
8  print '------------------' # end of table line (after loop)
```

WHILE loops (cont.)

Remark

Do not forget the colon (`:`) at the end of the `while` line

```
1 ...
2 while C <= 40:                    # loop heading with condition
3   ...
4 ...
```

- This colon is essential as it marks the beginning of the indented block of statements inside the loop

Remark

There are other similar program constructions in Python where there is a heading ending with colon, followed by an indented block of statements

WHILE loops (cont.)

It is necessary to understand what is going on in a program

- One should be able to **simulate a program by 'hand'**

WHILE loops (cont.)

```
1  print '------------------' # table heading
2
3  C = -20                   # start value for C
4  dC = 5                    # increment of C in loop
5
6  while C <= 40:            # loop heading with condition
7   F = (9.0/5)*C + 32       # 1st statement inside loop
8   print C, F               # 2nd statement inside loop
9   C = C + dC               # 3rd statement inside loop
10
11 print '------------------' # end of table line (after loop)
```

First, we define a start value for the sequence of Celsius temperatures

```
1  C = -20
2  dC = 5
```

We also define the increment `dC` to be added to `C` inside the loop

Then we enter the loop condition `C <= 40`

- The first time `C` is `-20`, `C <= 40` (equivalent to $C \leq 40$) is true

Condition is true, we enter the loop and execute all indented statements

WHILE loops (cont.)

```
1  print '------------------' # table heading
2  C = -20                   # start value for C
3  dC = 5                    # increment of C in loop
4
5  while C <= 40:            # loop heading with condition
6   F = (9.0/5)*C + 32       # 1st statement inside loop
7   print C, F               # 2nd statement inside loop
8   C = C + dC               # 3rd statement inside loop
9
10 print '------------------' # end of table line (after loop)
```

- We compute `F` corresponding to the current `C` value (-20)
- We print temperatures (`print C, F`, no formatting)
- We increment `C` (-20) by `dC` (5)

Thereafter, we enter the second pass in the loop

WHILE loops (cont.)

First we check the condition: `C` is now `-15` and `C <= 40` is still true

```
1  print '------------------' # table heading
2  C = -20                   # start value for C
3  dC = 5                    # increment of C in loop
4
5  while C <= 40:            # loop heading with condition
6   F = (9.0/5)*C + 32       # 1st statement inside loop
7   print C, F               # 2nd statement inside loop
8   C = C + dC               # 3rd statement inside loop
9
10 print '------------------' # end of table line (after loop)
```

We execute the statements in the indented loop block, concluding with `C` equal `-10`, which is less than or equal to `40`, we re-execute the block

- This procedure is repeated until `C` is updated from `40` to `45`
- When we test `C <= 40`, the condition is no longer true,
- The loop is thus terminated

WHILE loops (cont.)

```
1  print '------------------' # table heading
2  C = -20                   # start value for C
3  dC = 5                    # increment of C in loop
4
5  while C <= 40:            # loop heading with condition
6   F = (9.0/5)*C + 32       # 1st statement inside loop
7   print C, F               # 2nd statement inside loop
8   C = C + dC               # 3rd statement inside loop
9
10 print '------------------' # end of table line (after loop)
```

We proceed with the next statement with the same indentation of the `while statement`, which is the final `print statement` in the example

WHILE loops (cont.)

Remark

```
1 C = C + dC
```

Mathematically, the statement is wrong, but it is valid computer code

- We evaluate the expression on the RHS of the equality sign and let then the variable on the LHS 'refer' to the result of this evaluation

`C` and `dC` are `int objects`, operation `C+dC` returns a new `int object`

- The assignment `C = C + dC` bounds it to the name `C`

Before this assignment, `C` was already bound to an `int object`, and this object is automatically destroyed when `C` is bound to the new object

- There are no longer names (variables) referring to the old object

WHILE loops (cont.)

Remark

Incrementing the value of a variable is often done in computer codes

- There is short-hand notation for this and related operations

```
1 C += dC    # equivalent to C = C + dC
2
3 C -= dC    # equivalent to C = C - dC
4
5 C *= dC    # equivalent to C = C*dC
6
7 C /= dC    # equivalent to C = C/dC
```

Boolean expressions
WHILE loops

Boolean expressions

Condition `C <= 40` returned either true (`True`) or false (`False`)

Example

Other comparisons are also useful and commonly used

```
1 C == 40    # C equals 40
2 C != 40    # C does not equal 40
3 C >= 40    # C is greater than or equal to 40
4 C >  40    # C is greater than 40
5 C <  40    # C is less than 40
```

Not only comparisons between numbers can be used as conditions

- Any expression with boolean (`True` or `False`) value can be used
- Such expressions are known as **logical** or **boolean expressions**

Boolean expressions (cont.)

The keyword `not` can be inserted in front of a boolean expression

- It changes its value from `True` to `False`, or `False` to `True`

Example

To evaluate `not C == 40`, we first check `C == 40` as if `not (C == 40)`

- For `C = 1`, the statement `C == 40` is `False`
- `not` changes the value from `False` into `True`

If `C == 40` were `True`, `not C == 40` would be `False`

Remark

It is easier to read `C != 40` than `not C == 40`

- The two boolean expressions are equivalent

Boolean expressions (cont.)

Boolean expressions can be combined with `and` and/or `or`

- The goal is to form new compound boolean expressions

Example

```
1 while x > 0 and y <= 1:
2  print x, y
```

Boolean expressions (cont.)

Definition

Let `cond1` and `cond2` be two expressions: Valued either `True` or `False`

- The compound boolean expression `(cond1 and cond2)` is `True` only if both the conditions `cond1` and `cond2` are `True`
- The compound boolean expression `(cond1 or cond2)` is `True` only if at least one condition, `cond1` or `cond2`, is `True`

Boolean expressions (cont.)

Example

```
>>> x = 0; y = 1.2

>>> x >= 0 and y < 1
    False

>>> x >= 0 or y < 1
    True

>>> x > 0 or y > 1
    True

>>> x > 0 or not y > 1
    False

>>> -1 < x <= 0                                  # -1 < x and x <= 0
    True
```

Boolean expressions (cont.)

Example

```
>>> x = 0; y = 1.2

>>> not (x > 0 or y > 0)
    False
```

`not` applies to the value of the boolean expression inside parentheses:

- `x > 0` is `False`, `y > 0` is `True`, so the combined expression with `or` is `True`, and `not` turns the value to `False`

Boolean expressions (cont.)

Remark

Commonly used boolean values in Python are classic `True` and `False`

- We can also use `0` (`False`) and any non-zero integer (`True`)

Remark

All objects in Python can be evaluated in a boolean sense:

- All are `True` except `False` itself, zero numbers, and empty strings, lists, and dictionaries

Boolean expressions (cont.)

Example

```
>>> s = 'some string'                    # some string
>>> bool(s)
    True

>>> s = ''                               # empty string
>>> bool(s)
    False

>>> L = [1, 4, 6]                     # some list (soon)
>>> bool(L)
    True

>>> L = []                               # empty list
>>> bool(L)
    False

>>> a = 88.0                             # a scalar
>>> bool(a)
    True

>>> a = 0.0                               # a zero
>>> bool(a)
    False
```

Summation
WHILE loops

Summation

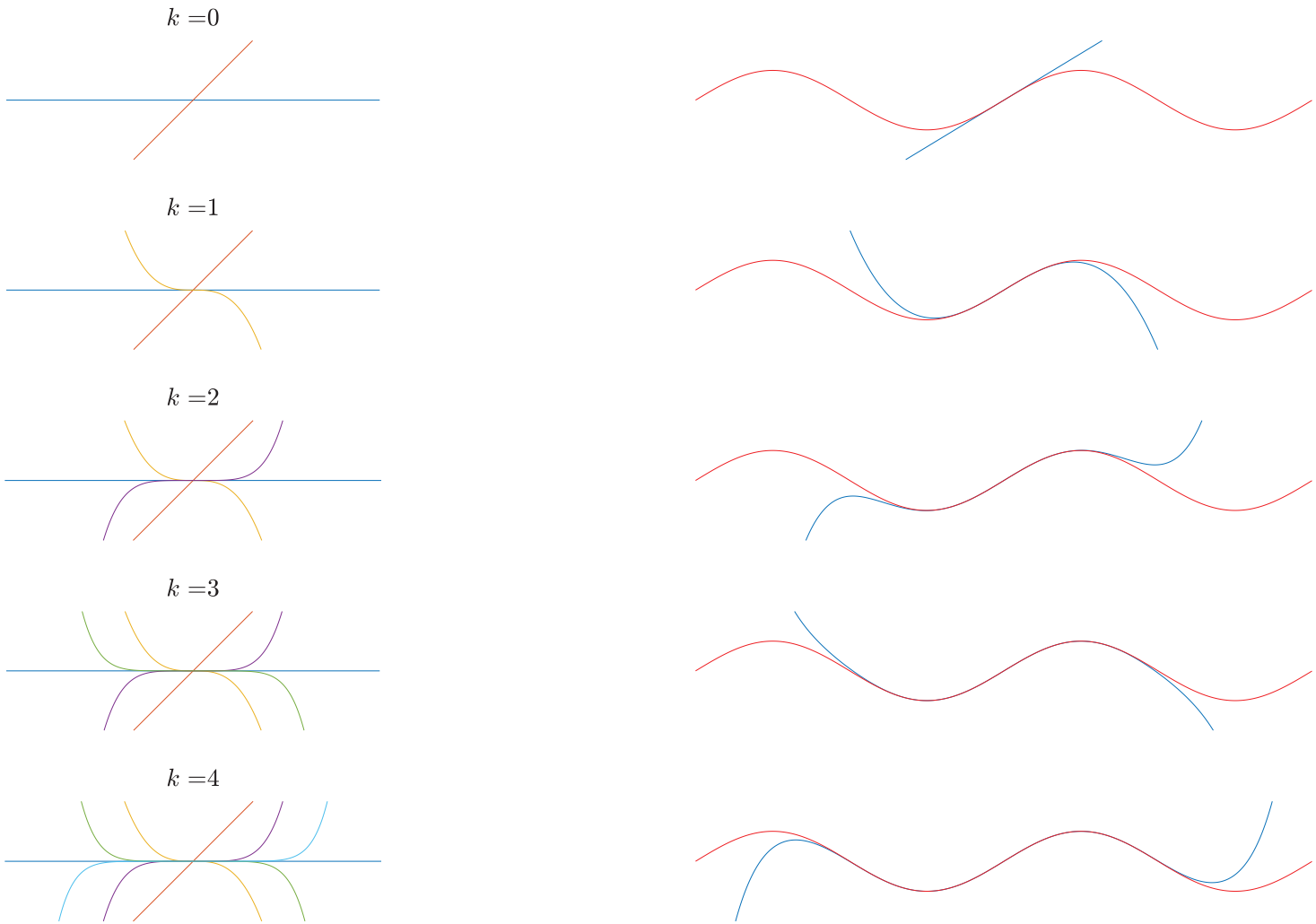
Example

Power series for sine: Approximate the sine function using a polynomial

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots = \sum_{k=0}^{\infty} \frac{(-1)^k x^{2k+1}}{(2k+1)!} \quad (1)$$

$3! = 3 \cdot 2 \cdot 1$, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, $7! = \cdots$, $\cdots$ are factorial expressions

Summation (cont.)



Summation (cont.)

Remark

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

- An infinite number of terms would be needed for equality to hold

With a finite number of terms, we obtain an approximation which is well suited for computation (only powers and the four arithmetic operations)

- Say, we want to compute for powers up to $N = 25$

Typing each term is a tedious job, task should be automated by a loop

Summation (cont.)

$$\sin(x) \approx \underbrace{x}_{\texttt{s(k=1)}} \underbrace{- \frac{x^3}{3!}}_{\texttt{s(k=3)}} \underbrace{+ \frac{x^5}{5!}}_{\texttt{s(k=5)}} \underbrace{- \frac{x^7}{7!}}_{\texttt{s(k=7)}} + \cdots + \underbrace{\frac{x^N}{N!}}_{\texttt{s(k=N)}}$$

To compute the summation by a `while loop` in Python we need

- A **counter**, `k`, that runs through odd numbers from `1` up to some maximum power `N` ($1, 3, 5, \cdots, N$)
- A **summation variable**, say `s`, that accumulates the terms, one at a time as they get computed

At each pass of the loop, compute a new term and add it to `s`

Since the sign of each term alternates, we use a variable `sign`

- It changes between `-1` and `+1` at each pass of the loop

Summation (cont.)

Remark

- `math.factorial(k)` can be used to compute $k!$ for some k

$$k! = k(k-1)(k-2)\cdots 2 \cdot 1$$

Summation (cont.)

$$\sin(x) \approx \underbrace{x}_{\texttt{s(k=1)}} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots + \frac{x^N}{N!}$$

```
1  x = 1.2                                    # assign some value
2  N = 25                                   # maximum power in sum
3
4  k = 1
5  s = x
6  sign = 1.0
7
8  import math
9
10 while k < N:
11  sign = - sign
12  k = k + 2
13
14  term = sign*x**k/math.factorial(k)
15
16  s = s + term
17 print 'sin(%g) = %g (approximation with %d terms)' % (x, s, N)
```

The loop is first entered, `k = 1 < 25 = N` (`1 < 25` implies `k < N`)

- The statement is `True`, we enter the loop block

Summation (cont.)

In the block, `sign = -1.0`, `k = 3`, `term = -1.0*x**3/(3*2*1))`

- `s = x - x**3/6` (equals to computing the first two terms)

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots + \frac{x^N}{N!}$$

```
1  x = 1.2                                          # assign some value
2  N = 25                                         # maximum power in sum
3
4  k = 1; s = x; sign = 1.0
5
6  import math
7
8  while k < N:
9   sign = - sign
10  k = k + 2
11
12  term = sign*x**k/math.factorial(k)
13
14  s = s + term
15 print 'sin(%g) = %g (approximation with %d terms)' % (x, s, N)
```

- Note that `sign` is `float` (always a `float` divided by an `int`)

Summation (cont.)

$$\sin(x) \approx \underbrace{x - \frac{x^3}{3!}}_{\texttt{s(k=3)}} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots + \frac{x^N}{N!}$$

```
1  x = 1.2                                          # assign some value
2  N = 25                                         # maximum power in sum
3
4  k = 1; s = x; sign = 1.0
5
6  import math
7
8  while k < N:
9   sign = - sign
10  k = k + 2
11
12  term = sign*x**k/math.factorial(k)
13
14  s = s + term
15 print 'sin(%g) = %g (approximation with %d terms)' % (x, s, N)
```

Then we test the loop condition: `3 < 25` is `True`, we re-enter the loop

- `term = + 1.0*x**5/math.factorial(5)` (third term in the sum)

Summation (cont.)

At some point, `k` is updated to from `23` to `25` inside the loop

- The loop condition becomes `25 < 25`, `False`
- Then the program jumps out the loop block

The `print statement` (indented as the `while statement`)

Lists
Loops and lists

Lists

Up to now a variable has contained a single number

- Often numbers are naturally grouped together

Example

For example, all degree Celsius values in the first column of the table

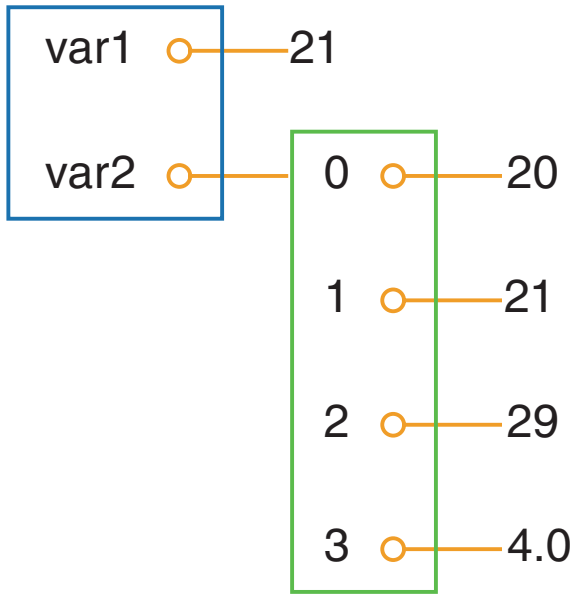
- They could be conveniently stored together as a group

A **list** can be used to represent such group of numbers

Lists (cont.)

With a variable that refers to the list, we can work with the group as a whole at once, but we can also access individual elements of the group

The difference between an `int object` and a `list object`



- `var1` refers to an `int object` with value `21` (from statement `var1 = 21`)
- `var2` refers to a `list object` with value `[20, 21, 29, 4.0]` three `int objects` and one `float object` (from `var2 = [20, 21, 29, 4.0]`)

Remark

A `list object` can contain an ordered sequence of arbitrary objects

Basic operations
Lists

Basic operations

To create a `list object` with all the numbers from the first column in the temperature table, we type them between square brackets

- Inside, the elements are separated by commas

Example

```
1 C = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
```

Variable `C` refers to a `list object` holding 13 `list elements`

- In this case, all `list elements` are `int objects`

Basic operations (cont.)

Each element in a `list object` is always associated with a **list index**

- The `list index` reflects the position of the elements in the list
- First element has `list index 0`, the second has `list index 1`, ...

Example

```
1 C = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
2 #      0    1    2   3  4  5   6   7   8   9  10  11  12
```

In list `C` there are `13 list indices`, starting with `0` and ending with `12`

- To access the `list element` with `list index 3` (it is to the fourth element in the list), we type `C[3]`
- `C[3]` refers to an `int object`, value `-5`

Basic operations (cont.)

List elements can be deleted from list objects
List elements can be inserted to list objects

- The functionality for these tasks are built into the list object and are accessed by a **dot notation**

C.append(v) appends a new element v to the end of the list
C.insert(i,v) inserts a new element v in position number i

Example

```
>>> C = [-10, -5, 0, 5, 10, 15, 20, 25, 30]       # create list
    #       0   1  2  3   4   5   6   7   8

>>> C.append(35)                  # add new element 35 at the end

>>> C                                                # view list C
    [-10, -5, 0, 5, 10, 15, 20, 25, 30, 35]
    #  0   1  2  3   4   5   6   7   8   9
```

The number of elements in a list is accessed by len(C)

Basic operations (cont.)

`List objects` can be added to each other, to join them back to front

Example

```
>>> C
    [-10, -5, 0, 5, 10, 15, 20, 25, 30, 35]
    #   0   1  2  3   4   5   6   7   8   9

>>> C = C + [40, 45]           # extend C at the end
>>> C
    [-10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
    #   0   1  2  3   4   5   6   7   8   9  10  11
```

The result of `C + [40,45]` is a new `list object`, assigned to `C`

Basic operations (cont.)

Remark

Note that what adding lists means is up to the `list object` to define

- Not surprisingly, addition of the two lists `C` and `[40, 45]` is defined as '*appending the second list to the first list*'
- With techniques of class programming it is possible to create own objects and define (if desired) what it means to add such objects

Basic operations (cont.)

`List elements` can be inserted anywhere in an existing list object

Example

```
>>> C
    [-10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
    #  0   1  2  3   4   5   6   7   8   9  10  11

>>> C.insert(0, -15)        # insert new element -15 as index 0

>>> C
    [-15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
    #  0    1   2  3  4   5   6   7   8   9  10  11  12
```

Basic operations (cont.)

Command `del C[i]` is used to remove element with index `i` from list `C`

- After removal: List has changed, `C[i]` refers to another element

Example

```
>>> C
    [-15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
    #  0    1   2  3  4   5   6   7   8   9  10  11  12

>>> del C[2]                                  # delete 3rd element
>>> C
    [-15, -10, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
    #  0    1  2  3   4   5   6   7   8   9  10  11

>>> del C[2]                          # delete what is now 3rd element
>>> C
    [-15, -10, 5, 10, 15, 20, 25, 30, 35, 40, 45]
    #  0    1  2   3   4   5   6   7   8   9  10

>>> len(C)                                        # length of list
    11
```

Basic operations (cont.)

Command `C.index(10)` returns the index corresponding to first element with value `10` (4th element in sample list, with index `3`)

Example

```
>>> C
    [-15, -10, 5, 10, 15, 20, 25, 30, 35, 40, 45]
    #  0    1  2   3   4   5   6   7   8   9  10

>>> C.index(10)                  # find index for an element (10)
    3
```

Basic operations (cont.)

To check if an object with value `10` is present as element in some list `C`

- It is possible to use the boolean expression `10 in C`

Example

```
>>> C
    [-15, -10, 5, 10, 15, 20, 25, 30, 35, 40, 45]

>>> 10 in C                                  # is 10 an element in C?
    True
```

Basic operations (cont.)

Python allows negative indices, which leads to indexing from the right

- `C[-1]` is the last element of list `C`
- `C[-2]` is the element before `C[-1]`
- `C[-3]` is the element before `C[-2]`
- ... and so forth

Example

```
>>> C
    [-15, -10,  5, 10, 15, 20, 25, 30, 35, 40, 45]
    #-11  -10  -9  -8  -7  -6  -5  -4  -3  -2  -1

>>> C[-1]                          # view the last list element
    45

>>> C[-2]                       # view the next last list element
    40
```

Basic operations (cont.)

Building lists by typing all elements separated by commas is tedious

- Such process that can easily be automated by a loop

Example

Build a list of degrees from -50 to $+200$ in steps of 2.5 degrees

```
1 C = []
2 C_value = -50
3 C_max = 200
4
5 while C_value <= C_max:
6  C.append(C_value)
7  C_value += 2.5
```

First, an empty list (`[]`), then a `WHILE-loop` to append elements

Basic operations (cont.)

There is a syntax for creating variables that refer to the list elements

- List a sequence of variables on the lhs of an assignment to a list

Example

```
1  >>> somelist = ['book.tex', 'book.log', 'book.pdf']
2  >>> texfile, logfile, pdf = somelist
3  >>> texfile
4      'book.tex'
5
6  >>> logfile
7      'book.log'
8
9  >>> pdf
10     'book.pdf'
```

The number of variables must match the number of lists's elements

Basic operations (cont.)

Remark

Some list operations are reached by dot notation (`C.append(e)`), while others requires the `list object` as **argument** to a **function** (`len(C)`)

Though `C.append` behaves like a function, it is a function reached thru a `list object`, we say that `append` is a **method** in the `list object`

There are no strict rules in Python on whether a functionality of an object should be reached through a method or a function

FOR loops
Lists

FOR loops

When the data are collected in a list, we usually want to perform the same operation on each element in the list

- We need to go through all list elements

Computer languages have a special construct for doing this conveniently

- In Python and other languages this construct is called a `FOR-loop`

FOR loops (cont.)

Example

`for C in degrees` construct creates a loop over elements in `degrees`

```
1 degrees = [0, 10, 20, 40, 100]
2
3 for C in degrees:
4  print 'list element:', C
5 print 'The degrees list has', len(degrees), 'elements'
```

- At each pass, variable `C` refers to an element in the list, starting with `degrees[0]`, proceeding with `degrees[1]`, ... and so on
- Looping ends with the last element `degrees[n-1]`
  (`n` is the number of list elements, `len(degrees)`)

FOR loops (cont.)

```
1 degrees = [0, 10, 20, 40, 100]
2
3 for C in degrees:
4
5  print 'list element:', C
6
7 print 'The degrees list has', len(degrees), 'elements'
```

The `FOR-loop` specification ends with a colon (`:`)

After the `:` comes a `block of statements` using the current element

- Each statement in the block must be indented
- (As with `WHILE-loops`)

The fist statement with the same indentation of the `for statement` is executed as the loop is terminated

FOR loops (cont.)

To get all details of the program, follow the execution flow by hand

```
1 degrees = [0, 10, 20, 40, 100]
2
3 for C in degrees:
4
5  print 'list element:', C
6
7 print 'The degrees list has', len(degrees), 'elements'
```

We first define a list, `degrees` consisting of `5` elements

- Then, we enter the `FOR-loop`

In the first pass, `C` refers to the first element of `degrees`

- `degrees[0]`, an `int object` holding value `0`

We print '`list element:`' and the current `C` value (`0`)

No more statements in the block, proceed to next pass

FOR loops (cont.)

```
1 degrees = [0, 10, 20, 40, 100]
2
3 for C in degrees:
4
5  print 'list element:', C
6
7 print 'The degrees list has', len(degrees), 'elements'
```

- `C` then refers to `degrees[1]`, an `int object` with value `10`, the output now prints `10` after the text
- We proceed with `C` as `20`, `40`, until `C` is `100`

After printing `list element:` with value `100`, we go to the statement after the indented `loop block`, which prints the number of list elements

FOR loops (cont.)

```
1 degrees = [0, 10, 20, 40, 100]
2
3 for C in degrees:
4  print 'list element:', C
5 print 'The degrees list has', len(degrees), 'elements'
```

Executing the code returns the output

```
1 list element: 0
2 list element: 10
3 list element: 20
4 list element: 40
5 list element: 100
6 The degrees list has 5 elements
```

Example

With knowledge of lists and FOR-loops over elements in lists, we write a program to collect all degrees Celsius in the table in a list Cdegrees

- Then, a FOR-loop to compute/print corresponding Fahrenheits

```
1 Cdegrees = [-20,-15,-10,-5,0,5,10,15,20,25,30,35,40]
2
3 print '   C      F'
4
5 for C in Cdegrees:
6  F = (9.0/5)*C + 32
7  print '%5d %5.1f' % (C, F)       # Print C, F would use default
8                                   # format, each number would be
9                                   # separated by a blank
```

```
1   C      F
2 -20   -4.0
3 -15    5.0
4
5 ...  .....
6
7  35   95.0
8  40  104.0
```

Alternative implementations

Loops and lists

Alternative implementations

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

- We explore possible constructs and programs to store numbers in lists and print out tables

WHILE loops as FOR loops

Alternative implementations

WHILE loops as FOR loops

Definition

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

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

becomes

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

Example

Printout of the Celsius-Fahrenheit table of temperatures

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

```
1 Cdegrees = [-20,-15,-10,-5,0,5,10,15,20,25,30,35,40]
2 print '   C      F'
3 for C in Cdegrees:
4  F = (9.0/5)*C + 32
5  print '%5d %5.1f' % (C, F)
```

```
1 C = -20
2 dC = 5
3 while C <= 40:
4  F = (9.0/5)*C + 32
5  print C, F
6  C = C + dC
```

Range construction
Alternative implementations

Range construction

It is tedious to manually type the many elements in `Cdegrees`

- We should use a loop to automate the list construction

The **range construction** is a particularly useful tool for such a task

Definition

`range(n)` generates integers in $[0, n-1]$, integer n is not included

- `0`, `1`, `2`, `...`, `n-1`

`range(start, stop, step)` generates a sequence of integers

- `start`, `start+step`, `start+2*step` up to `stop` (not included)

`range(start, stop)` is the same as `range(start, stop, 1)`

Range construction (cont.)

Example

- `range(2, 8, 3)` returns `2` and `2+(1*3)=5` (but not `8 = 2+(2*3)`)
- `range(1, 11, 2)` returns `1`, `3 = 1+(1*2)`, `5 = 1+(2*2)`, `7 = 1+(3*2)`, `9 = 1 + (4*2)`

Range construction (cont.)

A FOR-loop over integers generated by range

```
1 for i in range(start, stop, step):
2  ...
```

Example

We use it to create a list Cdegrees with values [-20,-15,...,40]

```
1 Cdegrees = []
2 for C in range(-20, 45, 5):
3  Cdegrees.append(C)
4
5 # or just
6 Cdegrees = range(-20, 45, 5)
```

Note that, to include 40, the upper limit must be greater than 40

Range construction (cont.)

Example

Suppose we want to create `Cdegrees` as `[-10,-7.5,-5,...,40]`

- This time we cannot use `range` directly, because `range` can only create integers and we have decimal degrees

In this more general case, we introduce an **integer counter** `i` and generate the `C` values by $C = -10 + i \cdot 2.5$, for $i = 0, 1, 2, \ldots, 20$

```
1 Cdegrees = []
2
3 for i in range(0, 21):
4  C = -10 + i*2.5
5  Cdegrees.append(C)
```

FOR loops with list indexes

Alternative implementations

FOR-loops with list indexes

Instead of iterating over a list directly with the usual construction

```
1 for element in somelist:
2  ...
```

we can iterate over list indices and index the list inside the loop

```
1 for i in range(len(somelist)):
2  element = somelist[i]
3  ...
```

- `len(somelist)` returns the length of `somelist`
- Indices start at `0`, the largest legal index is `len(somelist)-1`
- `range(len(somelist))` gives indices `0`, `1`, `...`, `len(somelist)-1`

FOR loops with list indexes (cont.)

Iterating over loop indices is useful when we need to process two lists

Example

Create lists `Cdegrees` and `Fdegrees`, then make a list to write a table

- The table must have `Cdegrees` and `Fdegrees` as its columns

```
1  n = 21
2  C_min = -10; C_max = 40                    # min and max C
3  dC = (C_max - C_min)/float(n-1)            # increment in C
4
5  Cdegrees = []                              # builds the C list
6  for i in range(0, n):
7   C = -10 + i*dC
8   Cdegrees.append(C)
9
10 Fdegrees = []                              # builds the F list
11 for C in Cdegrees:
12  F = (9.0/5)*C + 32
13  Fdegrees.append(F)
14
15 for i in range(len(Cdegrees)):             # builds the joint table
16  C = Cdegrees[i]
17  F = Fdegrees[i]
18  print '%5.1f %5.1f' % (C, F)
```

Iterating over a loop index is convenient in the final list

FOR loops with list indexes (cont.)

Instead of appending new elements to the lists, we can start with lists of correct size, containing zeros, and index the lists to fill in actual values

Definition

To create a list of length `n` consisting of zeros

```
1 somelist = [0]*n
```

FOR loops with list indexes (cont.)

Example

```
1  C_min = -10
2  C_max = 40
3  dC = (C_max - C_min)/float(n-1)                    # increment in C
4
5  Cdegrees = [0]*n                          # must be of correct length
6  for i in range(len(Cdegrees)):
7   Cdegrees[i] = -10 + i*dC
8
9
10 Fdegrees = [0]*n                          # must be of correct length
11 for i in range(len(Cdegrees)):
12  Fdegrees[i] = (9.0/5)*Cdegrees[i] + 32
13
14 for i in range(len(Cdegrees)):
15  print '%5.1f %5.1f' % (Cdegrees[i], Fdegrees[i])
```

Changing list elements
Alternative implementations

Changing list elements

We want to change `Cdegrees` by adding `5` (degrees) to its elements

```
1 for i in range(len(Cdegrees)):
2  Cdegrees[i] += 5
```

Variable `c` can only be used to read list elements, not change them

```
1 ...
2
3 c = Cdegrees[0]       # automatically done in the for statement
4 c += 5
5
6 c = Cdegrees[1]       # automatically done in the for statement
7 c += 5
8
9 ...
```

Remark

To change a list element, must used an assignment of the form

```
1 Cdegrees[i] = ...                  # changes the i-th list element
```

List comprehension
Alternative implementations

List comprehension

Running through a list and for each element create a new element in another list is a frequently encountered task

- Python has a special compact syntax for this
- **List comprehension**

Definition

The general syntax for `list comprehension` is summarised as

```
1 newlist = [E(e) for e in list]
```

where `E(e)` is an `expression` involving element `e` of list `list`

List comprehension (cont.)

Example

```
1 Cdegrees = [-5+i*0.5 for i in range(n)]
2
3 Fdegrees = [(9.0/5)*C+32 for C in Cdegrees]
4
5 C_plus_5 = [C+5 for C in Cdegrees]
```

Travessing multiple lists
Alternative implementations

Travessing multiple lists

We may use `Cdegrees` and `Fdegrees` lists 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 the integer indices

- So that we can index both lists

Example

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

Travessing multiple lists (cont.)

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

Definition

As an alternative to the loop over indices, Python offers a special syntax

```
1 for e1, e2, e3, ... in zip(list1, list2, list3, ...):
2  # work with element e1 from list1, element e2 from list2, ...
```

The `zip` function turns *n* lists (`list1`, `list2`, `...`) into a single list of *n*-tuples, in which each *n*-tuple (`e1`, `e2`, `...`) has its first element (`e1`) from the first list (`list1`), second element `e2` from second list `list2`, ...

Note: The loop stops when the end of the shortest list is reached

Travessing multiple lists (cont.)

Example

```
1 table = []
2 for C, F in zip(Cdegrees, Fdegrees):
3  print '%5d %5.1f' % (C, F)
```

```
1 table = [[C,F] for C,F in zip(Cdegrees,Fdegrees)]
```

Nested lists
Loops and lists

Nested lists

Nested lists are `list objects` in which `list elements` can be `list objects` themselves

Examples motivate the need for nested lists

- We shall illustrate some basic operations

A table as a row or column list

Nested lists

A table as a row or column list

In our table of data we used one separate list for each table column

With *n* columns, we need *n* `list objects` to handle table data

- We think of a table as one entity, not a collection of *n* columns
- It would be natural to use one argument for the whole table

Achieved by using a **nested list**, each entry in the list is a list itself

A `table object` is a list of lists, either

- a list of the row elements of the table, or
- a list of the column elements of the table

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

Example

```
1 Cdegrees = range(-20, 41, 5)              # -20, -15, ..., 35, 40
2 Fdegrees = [(9.0/5)*C + 32 for C in Cdegrees]
3
4 table = [Cdegrees, Fdegrees]
```

This table is a list of two columns, and each column is a list of numbers

With `table[0]`, we access the first element in the table

- The `Cdegrees` list

`table[0][2]` is the third element in the first element

- `Cdegrees[2]`

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

For tabular data with rows and columns, usually the convention is

- The underlying data are a nested list, in which the first index counts the rows and the second index counts the columns

To obtain this, we must construct `table` as a list of `[C, F]` pairs

- The first index will then run over rows `[C, F]`

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

or, more compactly

```
1 table = [[C, F] for C, F in zip(Cdegrees, Fdegrees)]
```

This construction is based on looping through pairs `C` and `F`

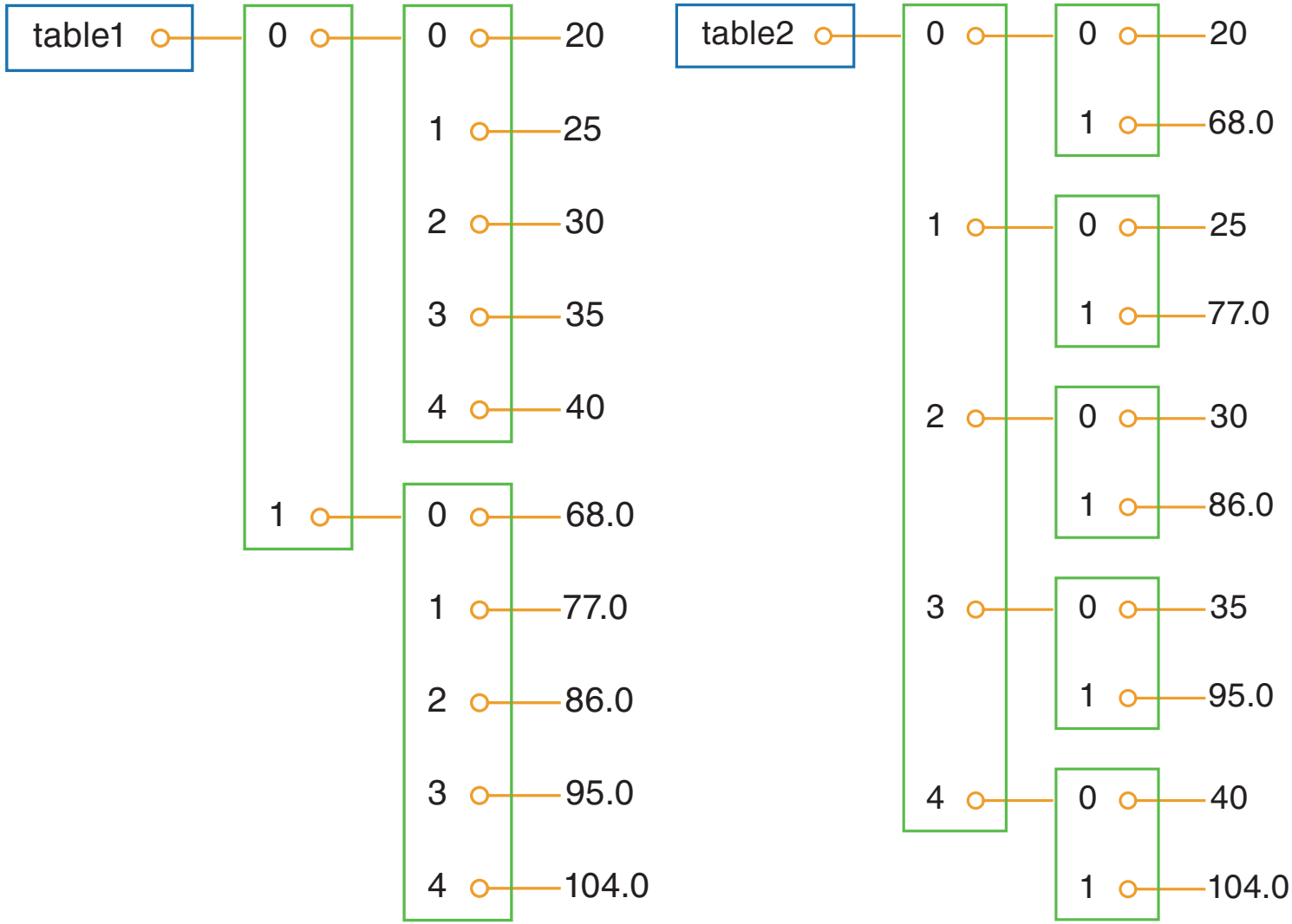
- At each pass, we create a list element `[C, F]`

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

```
1 table = [[C, F] for C, F in zip(Cdegrees, Fdegrees)]
```

- `table[1]` is the second element in `table`, a `[C, F]` pair
- With `table[1][0]`, we access the `C` value
- With `table[1][1]`, we access the `F` value

A list of columns and a list of pairs



The first index looks up an element in the outer list

- This element can be indexed with the second index

Printing objects
Nested lists

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

```
1 [[-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 `pprint module` offers a **pretty print** embellishing functionality

Example

```
1 import pprint
2 pprint.pprint(table)
```

```
1  [[-20,   -4.0],
2   [-15,    5.0],
3   [-10,   14.0],
4   [-5,    23.0],
5   [0,     32.0],
6   [5,     41.0],
7   [10,    50.0],
8   [15,    59.0],
9   [20,    68.0],
10  [25,    77.0],
11  [30,    86.0],
12  [35,    95.0],
13  [40,   104.0]]
```

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]]

>>> scitools.pprint2.pprint(somelist)
    [15.8, [0.2, 1.7]]

>>> # default output is '%g', change this to
>>> scitools.pprint2.float_format = '%.2e'
>>> scitools.pprint2.pprint(somelist)
    [1.58e+01, [2.00e-01, 1.70e+00]]
```

Printing objects (cont.)

The `pprint module` writes floating-point numbers with lots of digits

- To explicitly facilitate detection of round-off errors

Many find this type of output annoying and prefer the default output

- `scitools.pprint2` returns a conventional output

Printing objects (cont.)

Definition

`pprint` and `scitools.pprint2` `modules` have function `pformat`

- It works as `pprint`
- It returns a formatted string, rather than printing a string

```
1 s = pprint.pformat(somelist)
2 print s
```

The `print` statement prints like `pprint.pprint(somelist)`

Printing objects (cont.)

Tabular data such as those stored in `nested table lists` are not printed in a particularly pretty way by the `pprint module`

- The expected pretty output is two aligned columns

To produce such output, we will have to code the formatting

Example

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

- Print these in fixed width fields, using the `printf` syntax

```
1 for C, F in table:
2  print '%5d %5.1f' % (C, F)
```

Extracting sublists
Nested lists

Extracting sublists

Python has a syntax for extracting parts of a `list` structure

- **Sublists** or **slices**

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

```
>>> 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`

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

>>> A[:3]
    [2, 3.5, 8]
```

Extracting sublists (cont.)

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

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

>>> A[1:3]
    [3.5, 8]
```

`A[1:-1]` extracts all elements except the first and the last (recall that index `-1` refers to the last element), and `A[:]` refers to the whole list

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

>>> A[1:-1]
    [3.5, 8]

>>> A[:]
    [2, 3.5, 8, 10]
```

Extracting sublists (cont.)

```
[[-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]
```

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

```
>>> table[4:]
    [[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]]
```

Extracting sublists (cont.)

```
[[-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

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

`table[4:7]` makes a 3-element list `[[0,32.0],[5,41.0],[10,50.0]]`

- Slice `[0:2]` acts on it, picks its first two elements, indices `0` and `1`

Extracting sublists (cont.)

Sublists are always copies of the original list

Example

```
>>> l1 = [1, 4, 3]
>>> l2 = l1[:-1]
>>> l2
    [1, 4]

>>> l1[0] = 100
>>> l1 # l1 is modified
    [100, 4, 3]

>>> l2 # l2 is not modified
    [1, 4]
```

Remark

If you modify a sublist, the original list remains unaltered and *vice versa*

Extracting sublists (cont.)

Definition

`B == A` is `True` if all elements in `B` equal corresponding elements in `A`

- The test `B is A` is `True` if `A` and `B` are names for the same list

Example

```
>>> B = A[:]
>>> C = A

>>> B == A
    True

>>> B is A
    False

>>> C is A
    True
```

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

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

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

A `FOR-loop` does an equivalent job: `for C, F in table[6:11]:`

- `Cdegrees.index(10)` is the index of value `10` in the `Cdegrees` list
- `Cdegrees.index(10)` is the index of value `10` in the `Cdegrees` list

Travessing nested lists
Nested lists

Travessing nested lists

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

```
1 for C, F in table:
2  # process C and F
```

This is natural, when we know that `table` is a list of `[C, F]` lists

More general nested lists, where we do not necessarily know how many elements there are in each list element of the list, are handled differently

Travessing nested lists (cont.)

Example

Suppose a nested list `scores` recording scores of players in a game

- `scores[i]` holds the list of scores obtained by player number `i`

Different players have played the game a different number of times

- The length of `scores[i]` depends on `i`

```
1  scores = []
2
3  # score of player no. 0:
4  scores.append([12, 16, 11, 12])
5
6  # score of player no. 1:
7  scores.append([9])
8
9  # score of player no. 2:
10 scores.append([6, 9, 11, 14, 17, 15, 14, 20])
```

The list has three elements, each element corresponding to a player

Travessing nested lists (cont.)

Element number `g` in the list `scores[p]` corresponds to the score obtained in game number `g` played by player number `p`

- The length of lists `scores[p]` varies
- It equals `4`, `1`, and `8` for `p` equal `0`, `1`, and `2`, respectively

Travessing nested lists (cont.)

Remark

In the general case of `n` players, some may have played the game a large number of times, making `scores` potentially a big nested list

Example

The data initialised earlier can be written out as

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

How to traverse the list and put it in table format with well formatted columns?

- Each row corresponds to a player
- Columns correspond to scores

Travessing nested lists (cont.)

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

We may use two nested loops

- one for the elements in `scores`
- one 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

Travessing nested lists (cont.)

An index-based version, a trailing comma after '`print string`' is used

```
1 for p in range(len(scores)):
2  for g in range(len(scores[p])):
3   score = scores[p][g]
4   print '%4d' % score,
5  print
```

A `print` after the loop over `p` adds a new (empty) line after each row

With variables for iterating over the elements in `scores` and its sublists

```
1 for player in scores:
2  for game in player:
3   print '%4d' % game,
4  print
```

Travessing nested lists (cont.)

Definition

When we have nested lists with many indices

- `somelist [i1][i2][i3] ...`

To visit each element in the list, we use as many nested `FOR-loops` as there are indices

Travessing nested lists (cont.)

With four indices, iterating over integer indices looks like

```
1 for i1 in range(len(somelist)):
2  for i2 in range(len(somelist[i1])):
3   for i3 in range(len(somelist[i1][i2])):
4    for i4 in range(len(somelist[i1][i2][i3])):
5
6     value = somelist[i1][i2][i3][i4]
7     # work with value
```

The corresponding version iterating over sublists becomes

```
1 for sublist1 in somelist:
2  for sublist2 in sublist1:
3   for sublist3 in sublist2:
4    for sublist4 in sublist3:
5
6     value = sublist4
7     # work with value
```

Some list operations
Nested lists

Some list operations

Construct	Explaination
`a = []`	Initialise an empty string
`a = [1, 4.4, 'run.py']`	Initialise a list
`a.append(elem)`	Add `element`
`a + [1.3]`	Add two lists
`a.insert(i, e)`	Insert element `e` before index `i`
`a[3]`	Index a list element
`a[-1]`	Get last lists element
`a[1:3]`	Slide: Copy data to sublist
`del a[3]`	Delete an element
`a.remove(e)`	Remove an element with value `e`

Some list operations (cont.)

Construct	Explaination
`a.index('run.py')`	Index corresponding to element's value
`'run.py' in a`	Test if a value is in the list
`a.count(v)`	Count elements with value `v`
`len(a)`	Number of elements in list `a`
`min(a)`	The smallest element in list `a`
`max(a)`	The largest element in list `a`
`sum(a)`	Add all elements in `a`
`sorted(a)`	Return sorted version of `a`
`reversed(a)`	Return returned version of `a`
`b[3][0][2]`	Nested list indexing
`isinstance(a, list)`	`True` if `a` is a list
`type(a) is list`	`True` if `a` is a list

Tuples
Lists and loops

Tuples

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

```
1 >>> t = (2, 4, 6, 'temp.pdf')      # define a tuple with name t
```

Sometimes, we can even drop the parentheses

```
1 >>> for element in 'myfile.txt', 'urfile.txt', 'herfile.txt':
2     ... print element,
3     ...
4
5     myfile.txt yourfile.txt herfile.txt
```

Remark

The `FOR-loop` is over a tuple: A comma-separated sequences of objects, even without the parentheses, become automatically `tuple objects`

Tuples (cont.)

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

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

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

The comma suppresses the final newline that a `print` command would add to the output `string object`

Remark

This is how to make several `print` statements build up one output line

Tuples (cont.)

Many functionalities for lists are also available for tuples

```
>>> 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
```

Tuples (cont.)

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

```
>>> 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
```

Some methods for lists (like `index`) are not available for tuples

Tuples (cont.)

So why do we need tuples 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 frequently used in Python software that you will make use of, so you need to know this data type

There is also a fourth argument, important for a data-type called dictionaries, tuples can be used as keys in dictionaries, lists cannot