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 a new type of object, the **list objects**

- For storing and processing collections of data
- (with a specific order)

Loops and lists, with functions/routines and IF-tests (soon)

- The fundamental programming foundation

WHILE loops

Loops and lists

WHILE loops

Example

We are interested in printing out a temperature conversion 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
```

- Degree Celsius in the first column of the table
- Corresponding Fahrenheits in the second one

WHILE loops

The formula for converting C degrees Celsius to F degrees Fahrenheit

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

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 can repeatedly write the whole command

- (`c2f_table_repeat.py`)

```
C = -20; F = 9.0/5*C + 32; print C, F
C = -15; F = 9.0/5*C + 32; print C, F
C = -10; F = 9.0/5*C + 32; print C, F
C = -5;  F = 9.0/5*C + 32; print C, F
C = 0;   F = 9.0/5*C + 32; print C, F
C = 5;   F = 9.0/5*C + 32; print C, F
C = 10;  F = 9.0/5*C + 32; print C, F
C = 15;  F = 9.0/5*C + 32; print C, F
C = 20;  F = 9.0/5*C + 32; print C, F
C = 25;  F = 9.0/5*C + 32; print C, F
C = 30;  F = 9.0/5*C + 32; print C, F
C = 35;  F = 9.0/5*C + 32; print C, F
C = 40;  F = 9.0/5*C + 32; print C, F
```

We used three statements per line in the code

- For compacting the layout

WHILE loops (cont.)

We can run this program and show how the output looks like on screen

```
-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
```

Remark

The output of the code suffers from a rather primitive text formatting

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

WHILE loops (cont.)

```
C = -20; F = 9.0/5*C + 32; print C, F
C = -15; F = 9.0/5*C + 32; print C, F

  ...
  ...

C = 40;  F = 9.0/5*C + 32; print C, F
```

The major problem with this code is that identical statements are repeated

- It is boring and dumb to write repeated statements
- (Imagine many more C and F values in the table)

WHILE loops (cont.)

All computer languages have constructs to efficiently express repetition

- One of the ideas behind a computer is to automate repetitions

Such constructs are called **loops**

We have two variants in Python

⇝ **WHILE-loops**
⇝ **FOR-loops**

Most programs make an extensive use of loops

- It is fundamental to learn the concept

WHILE loops

WHILE loops

WHILE loops (cont.)

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

- It repeats as long as a some condition is verified (true)

To illustrate this loop, we use the temperature table

WHILE loops

Example

The task is to generate the rows of the table

- 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 they are incremented by 5

- This process is repeated, as long as $C \leq 40$

WHILE loops (cont.)

```
-20  -4.0
-15   5.0

...  ....
...  ....

 40 104.0
```

For each C value, we must first compute the corresponding F value

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

Then, we write out (print to screen) the two temperatures

For cosmetics, we would also like add a line of dashes (`--`)

- One above and one below the table

WHILE loops (cont.)

```
-20  -4.0
-15   5.0

...  ....
...  ....

 40 104.0
```

The list of tasks to be done can be summarised

```
❶ Print line with dashes
❷ Let $C = -20$
❸ WHILE $C \leq 40$:
    ⤳ Let $F = 9/5C + 32$
    ⤳ Print $C$ and $F$
    ⤳ Increment $C$ by $5$
❹ Print line with dashes
```

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

WHILE loops (cont.)

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

Converting a detailed algorithm into a functioning code is often easy

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

WHILE loops (cont.)

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

- It must be indented

```
print ’------------------’ # table heading

C = -20                    # start value for C
dC = 5                     # increment of C in loop

while C <= 40:             # loop heading with condition

 F = (9.0/5)*C + 32        # 1st statement inside loop
 print C, F                # 2nd statement inside loop
 C = C + dC                # 3rd statement inside loop

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 space
- (Usually, it is four space)

WHILE loops (cont.)

```
print '------------------' # table heading

C = -20                    # start value for C
dC = 5                     # increment of C in loop

while C <= 40:             # loop heading with condition

 F = (9.0/5)*C + 32        # 1st statement inside loop
 print C, F                # 2nd statement inside loop
 C = C + dC                # 3rd statement inside loop

print '------------------' # end of table line (after loop)
```

Consider the first statement with same indentation as the `while` line

- (Here, the final `print statement`)

This line marks the end of the loop

- It is executed after the loop

WHILE loops (cont.)

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

```
print '------------------'  # table heading
C = -20                     # start value for C
dC = 5                      # increment of C in loop

while C <= 40:              # loop heading with condition
 F = (9.0/5)*C + 32         # 1st statement inside loop
 print C, F                 # 2nd statement inside loop
 C = C + dC                 # 3rd statement inside loop
 print '------------------' # end of table line (after loop)
```

WHILE loops (cont.)

Remark

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

```
...

while C <= 40:                   # loop heading with condition
 ...
 ...

...                              # after the loop
```

The colon marks the beginning of the indented block of statements

- The colon marks the loop, it is essential

WHILE loops (cont.)

Remark

A heading ending with colon, followed by an indented block of statements

- There are other similar program constructions in Python

WHILE loops (cont.)

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

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

WHILE loops (cont.)

```
print '------------------' # table heading

C = -20                     # start value for C
dC = 5                      # increment of C in loop

while C <= 40:              # loop heading with condition
 F = (9.0/5)*C + 32         # 1st statement inside loop
 print C, F                 # 2nd statement inside loop
 C = C + dC                 # 3rd statement inside loop

print '------------------' # end of table line (after loop)
```

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

```
C = -20
dC = 5
```

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

WHILE loops (cont.)

```
print '-----------------' # table heading

C = -20                   # start value for C
dC = 5                    # increment of C in loop

while C <= 40:            # loop heading with condition
 F = (9.0/5)*C + 32       # 1st statement inside loop
 print C, F               # 2nd statement inside loop
 C = C + dC               # 3rd statement inside loop

print '-----------------' # end of table line (after loop)
```

Then, we enter/define the loop condition `C <= 40`

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

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

WHILE loops (cont.)

```
print '------------------' # table heading
C = -20                    # start value for C
dC = 5                     # increment of C in loop

while C <= 40:             # loop heading with condition
 F = (9.0/5)*C + 32        # 1st statement inside loop
 print C, F                # 2nd statement inside loop
 C = C + dC                # 3rd statement inside loop

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)
- (What's the value of `C`?)

Thereafter, we may enter the loop again

- The second pass

WHILE loops (cont.)

To decide whether to re-enter the loop, we must check condition `C <= 40`

- `C <= 40` is still true
- `C` is now `-15`

```
print '-------------------' # table heading
C = -20                    # start value for C
dC = 5                     # increment of C in loop

while C <= 40:             # loop heading with condition
 F = (9.0/5)*C + 32        # 1st statement inside loop
 print C, F                # 2nd statement inside loop
 C = C + dC                # 3rd statement inside loop

print '-------------------' # end of table line (after loop)
```

We execute the statements in the indented loop block

We conclude those computations with `C` equal `-10`

- It is less than or equal to `40`

We thus re-execute the block

WHILE loops (cont.)

-20, -15, -10, · · ·, 35, 40, · · ·

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

```
print '------------------' # table heading
C = -20                   # start value for C
dC = 5                    # increment of C in loop

while C <= 40:            # loop heading with condition
 F = (9.0/5)*C + 32       # 1st statement inside loop
 print C, F               # 2nd statement inside loop
 C = C + dC               # 3rd statement inside loop

print '------------------' # end of table line (after loop)
```

We proceed with the next statement, same indentationas `while statement`

⤳ We execute the final `print statement`

WHILE loops (cont.)

Remark

Consider the following statement used in the code

```
C = C + dC
```

Mathematically, the statement is wrong

- Yet, it is valid computer code

Computationally, we first evaluate the expression on RHS of equality sign

We then let variable on the LHS ‘refer’ to the result of this evaluation

WHILE loops (cont.)

```
C = C + dC
```

`C` and `dC` are `int objects`, the 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`

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/object is often done in computer codes

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

```
C += dC    # equivalent to C = C + dC
```

The idea can be extended to other operators

```
C -= dC    # equivalent to C = C - dC

C *= dC    # equivalent to C = C*dC

C /= dC    # equivalent to C = C/dC
```

Boolean expressions

WHILE loops

Boolean expressions

```
print '------------------' # table heading
C = -20                    # start value for C
dC = 5                     # increment of C in loop

while C <= 40:             # loop heading with condition
 F = (9.0/5)*C + 32        # 1st statement inside loop
 print C, F                # 2nd statement inside loop
 C = C + dC                # 3rd statement inside loop

print '------------------' # end of table line (after loop)
```

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

Boolean expressions (cont.)

There exist other comparisons are also useful and commonly used

```
C == 40   # C equals 40
C != 40   # C does not equal 40
C >= 40   # C is greater than or equal to 40
C >  40   # C is greater than 40
C <  40   # C is less than 40
```

Clearly, not only comparisons between numbers can be used to set conditions

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

Boolean expressions (cont.)

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

- It changes its value

⤳ (`True` to `False`)

⤳ (`False` to `True`)

Boolean expressions (cont.)

Example

Suppose that we want to evaluate the output of `not C == 40`

We first check `C == 40`, and then `not (C == 40)`

- For `C = 1`, the statement `C == 40` is `False`

⤳ `not` changes the value, `False` into `True`

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

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

- The two boolean expressions are equivalent

Boolean expressions (cont.)

As in math, Boolean expressions can be combined with `and` and/or `or`

- The goal is to form new, compound, boolean expressions

Example

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

Boolean expressions (cont.)

Definition

Let `cond1` *and* `cond2` *be two expressions*

- *Valued either* `True` *or* `False`

Consider the compound boolean expression `(cond1 and cond2)`

- *It is* `True` *only if both the conditions* `cond1` *and* `cond2` *are* `True`

The compound boolean expression `(cond1 or cond2)`

- *It 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
```

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

- `x > 0` is `False`, `y > 0` is `True`

The combined expression with `or` is `True`, and `not` turns the value to `False`

Boolean expressions (cont.)

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

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

All objects in Python can be evaluated in a boolean sense

- All objects 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

We can approximate the sine function using a polynomial

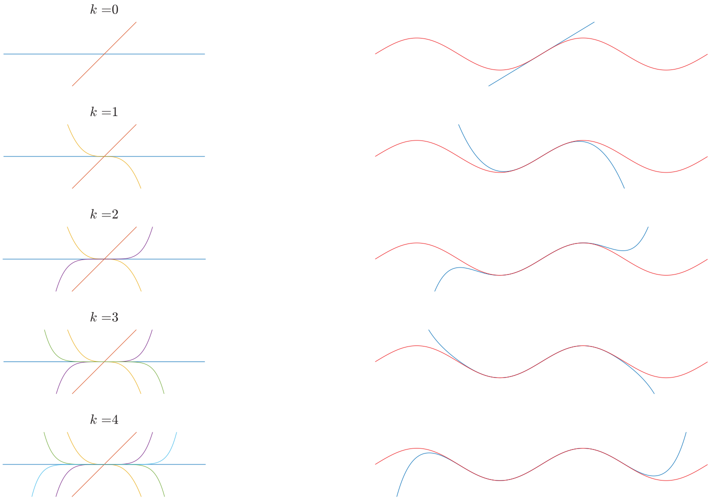
$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots = \sum_{k=0}^{\infty} \frac{(-1)^k x^{2k+1}}{(2k+1)!} \tag{1}$$

We used the factorial expressions

- $3! = 3 \cdot 2 \cdot 1$
- $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$
- $7! = 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$
- $\cdots$

Summation (cont.)

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



Summation (cont.)

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

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

With a finite number of terms, we obtain an approximation

The approximation is well suited for computation

- (powers and four arithmetic operations)

Summation (cont.)

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

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

- Typing each term is a tedious job

Clearly, this task should be automated by a loop

Summation (cont.)

We are interested in computing the summation by a `while loop` in Python

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

What do we need?

A **counter**, say `k`

- It runs through odd numbers from `1` up to some maximum power `N`
- $(1, 3, 5, \cdots, N)$

A **summation variable**, say `s`

- It accumulates the terms, one at a time as they get computed
- At each pass, we compute a new term and add it to `s`

Summation (cont.)

The sign of each term in the summation alternates

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

We use a **sign variable**, say `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.)

Let $x = 1.2$

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

```
x = 1.2                                                   #    assign some value
N = 25                                                    # maximum power in sum

k = 1                                                   # initialise the counter
s = x                                                   #     initialise the sum
sign = 1.0                                              #          set the sign

import math                                     # needed to access the factorial

while k < N:
 sign = - sign
 k = k + 2

 term = sign*x**k/math.factorial(k)

 s = s + term
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 holds `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)

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

```python
x = 1.2
N = 25

k = 1; s = x; sign = 1.0

import math

while k < N:
 sign = - sign                                                   # update sign
 k = k + 2                                                          # update k

 term = sign*x**k/math.factorial(k)                              # compute term

 s = s + term                                                 # updates the sum

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.)

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

```
x = 1.2                                                    # assign some value
N = 25                                                    # maximum power in sum

k = 1; s = x; sign = 1.0

import math

while k < N:
 sign = - sign
 k = k + 2

 term = sign*x**k/math.factorial(k)

 s = s + term
print 'sin(%g) = %g (approximation with %d terms)' % (x, s, N)
```

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

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

Summation (cont.)

```
while k < N:
 ...
 k = k + 2
 ...

print 'sin(%g) = %g (approximation with %d terms)' % (x, s, N)
```

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

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

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

Lists

Loops and lists

Lists

Up to now we considered variables that contained a single number

- Often numbers are naturally grouped together
- We have collections of numbers

Lists (cont.)

Example

All degree Celsius values in the first column of the temperature table

- They could be conveniently stored together as a group

```
-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
```

Lists (cont.)

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

⤳ A `list object`

Functionalities for examination and manipulation

Remark

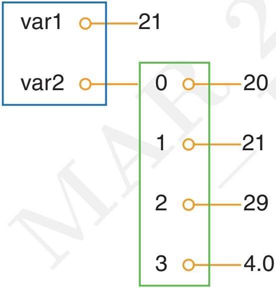
A `list object` can contain an ordered sequence of arbitrary objects

Lists (cont.)

Consider some variable that refers to some list

- ⤳ We can work with the group as a whole at once
- ⤳ We can access individual elements of the group

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



`var1` refers to an `int object`

- Value `21`
- (from statement `var1 = 21`)

`var2` refers to a `list object`

- Value `[20, 21, 29, 4.0]`
- Three `int objects`, one `float object`
- (from `var2 = [20, 21, 29, 4.0]`)

Basic operations

Lists

Basic operations

```
-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
```

Suppose that we are interested in creating a `list object`

- Numbers in the first column of a temperature table

We type each number individually between square brackets

- Inside, the elements are separated by commas

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

Basic operations (cont.)

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

Variable `C` is used to refer to a `list object`

- The object holds 13 `list elements`
- All `list elements` are `int objects`

Basic operations (cont.)

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

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

```
-20   # List index 0
-15   # List index 1
-10   # List index 2
 -5   # List index 3
  0   # List index 4
  5   # List index 5
 10   # List index 6
 15   # List index 7
 20   # List index 8
 25   # List index 9
 30   # List index 10
 35   # List index 11
 40   # List index 12
```

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

Basic operations (cont.)

Example

```
C = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
#      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`, we type `C[3]`

- (This is to the fourth element in the list)
- `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`

Functionalities for these tasks are built into the `list object`

- They are accessed by a **dot notation**

Basic operations (cont.)

Consider some list `C`

Function `C.append(v)` appends a new element `v` to the end of the list

Example

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


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


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

Basic operations (cont.)

Consider two (or more) `list objects`

`List objects` can be added to each other

- Addition (`+`) joins 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 existing list C
                                                 #      add list [40, 45]
                                                 #             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`

- New object is assigned to `C`

Basic operations (cont.)

Remark

The addition operation for list operands is defined by the `list object`

- The definition is '*append the second list to the first list*'
- (Not surprising!)

The techniques of class programming allow to create own object types

⤳ We can define (if desired) what it means to add such objects

Basic operations (cont.)

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

Consider some list `C`

Function `C.insert(i,v)` inserts a new element `v` in position number `i`

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
                                                #                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, original list has changed
- `C[i]` now refers to a different 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
```

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

Basic operations (cont.)

Command `C.index(10)` returns the index of the first element with value `10`

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
```

⤳ (4th element in sample list, with index `3`)

Basic operations (cont.)

We want to check if an object with value `10` is present as element in list `C`

- It is possible to use a 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, this corresponds 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

Basic operations (cont.)

Example

Suppose that we are interested in building a list of Celsius degree values

- -50 to $+200$
- Steps of 2.5

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

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

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

Basic operations (cont.)

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

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

Example

```
>>> somelist = [’book.tex’, ’book.log’, ’book.pdf’]

>>> texfile, logfile, pdf = somelist

>>> texfile
    ’book.tex’

>>> logfile
    ’book.log’

>>> pdf
    ’book.pdf’
```

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

Basic operations (cont.)

Remark

Some list operations are directly reached by dot notation

⤳ `C.append(e)`

Other requires the `list object` as **argument** to a **function**

⤳ `len(C)`

Though `C.append` behaves like a function, it is reached thru a `list object`

- We say that `append` is a **method** in the `list object`

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