Ch.3: Loops and lists

Joakim Sundnes[1,2]

[1]Simula Research Laboratory
[2]University of Oslo, Dept. of Informatics

Aug 31, 2023

0.1 Plan for 31 August

- Short quiz on topics from Tuesday
- Controlling output format (leftover from Tuesday)
- Exercise 1.12 from *Primer on Scientific Programming with Python*
- Loops and lists
  - The while loop
  - Boolean expressions (True/False)
  - The for loop
- Exercise 2.1 and 2.3 from *Primer on Scientific Programming with Python*
- More on loops and lists (most likely next week, partly self study):
  - range, zip, list comprehensions,...

0.2 Quiz question 1

Which of the following code segments are wrong (if any)? What is wrong?

Code 1

```
a = "Hello world"
a = 2
```

Code 2

```
pi = 3.14159
pi = 2 * pi
```

Code 3

```
b = x**2 + 3 * x + 1
x = 2.3
```

0.3 Answer to question 1

```
#code 1
a = "Hello world"
a = 2

#code 2
pi = 3.14159
pi = 2 * pi

#Code 3
b = x**2 + 3 * x + 1
x = 2.3
```

```
Terminal> python quiz1.py
Traceback (most recent call last):
  File "quiz1.py", line 10, in <module>
    b = x**2 + 3 * x + 1
NameError: name 'x' is not defined
```

In programming, variables must be defined before they are used (unlike mathematics).

0.4 Question 2

What are the types of the variables in the following code:

```
a = 2
b = 2.5
s = "hello"
t = a * s
```

0.5 Answer to question 2

```
a = 2
b = 2.5
s = "hello"
t = a * s

print('a is ', type(a), ' b is ', type(b), \
' s is ', type(s), ' t is ', type(t))
```

```
Terminal> python quiz2.py
a is  <class 'int'>  b is  <class 'float'>  s is  <class 'str'>  t is  <class 'str'>
```

0.6 Question 3

Which of these codes are wrong (if any)?

```
from math import sin, pi
x = sin(pi / 2)
```

```
import math
x = sin(pi / 2)
```

```
from math import *
x = sin(pi / 2)
```

0.7 Answer to question 3

The second import code is wrong. If you import a module in this way, all functions and variables from the module must be prefixed with the module name:

```
import math
x = math.sin(math.pi / 2)
```

0.8 Question 4 (discussion)

The Python module `cmath` is for computing with complex numbers, while `numpy` is a module for computing with arrays (many numbers at once).

Why is this code segment a bad idea:

```
from math import *
from numpy import *
from cmath import *

(...)
x = sin(pi / 2)
```

0.9 Answer to question 4

The three modules have many functions with identical names. If we import them like this it is very difficult to know which functions we use. When combining modules with potential name conflicts, we should use something like:

```
import math
import numpy
import cmath

(...)
x = math.sin(math.pi / 2)
```

0.10 Question 5 (discussion)

In Python, we can make and later change a variable like this:

```
a = 2
(...)
a = 0.5*2
```

In many other languages, we must write something like:

```
int a;
a = 2;
(...)
```

Python obviously saves some typing, but can you think of any potential problems with the Python way of defining variables?

0.11 Answer to question 5

This toy example illustrates one potential pitfall of dynamic typing:

```
#create a variable x0:
x0 = 10.0

#Do something useful...

#change x0:
xO = 14.0

#More important calculations with x0...
print('The value of x0 is ', x0)
```

0.12 Main topics of Chapter 3

- Using loops for repeating similar operations:
  - The `while` loop
  - The `for` loop
- Boolean expressions (True/False)
- Lists

0.13 Make a table of Celsius and Fahrenheit degrees

```
-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
```

How can a program write out such a table?

0.14 Making a table: the simple naive solution

We know how to make one line in the table:

```
C = -20
F = 9.0/5*C + 32
print(C, F)
```

We can just repeat these statements:

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

- Very boring to write, easy to introduce a typo
- When programming becomes boring, there is usually a construct that automates the writing!
- The computer is extremely good at performing repetitive tasks
- For this purpose we use *loops*

0.15 The while loop makes it possible to repeat similar tasks

A while loop executes repeatedly a set of statements as long as a boolean condition is true

```
while condition:
    <statement 1>
    <statement 2>
    ...
<first statement after loop>
```

- All statements in the loop must be indented!
- The loop ends when an unindented statement is encountered

0.16 Example 1: table with while loop

The while loop is a far more efficient way to make the Fahrenheit-Celcius table described on the previous slides.

Task: Given a range of Celsius degrees from -20 to 40, in steps of 5, calculate the corresponding degrees Fahrenheit and print both values to the screen.

0.17 The while loop for making a table

```
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              # last statement inside loop
print('------------------') # end of table line
```

0.18 The program flow in a while loop

Let us simulate the while loop by hand:

- First `C` is -20, $-20 \leq 40$ is true, therefore we execute the loop statements
- Compute `F`, print, and update `C` to -15
- We jump up to the `while` line, evaluate $C \leq 40$, which is true, hence a new round in the loop
- We continue this way until `C` is updated to 45
- Now the loop condition $45 \leq 40$ is false, and the program jumps to the first line after the loop - the loop is over

0.19 Boolean expressions are true or false

An expression with value true or false is called a boolean expression. Examples: $C = 40$, $C \neq 40$, $C \geq 40$, $C > 40$, $C < 40$.

```
C == 40  # note the double ==, C = 40 is an assignment!
C != 40
C >= 40
C >  40
C <  40
```

We can test boolean expressions in a Python shell:

```
>>> C = 41
>>> C != 40
True
>>> C < 40
False
>>> C == 41
True
```

0.20 Combining boolean expressions

Several conditions can be combined with and/or:

```
while condition1 and condition2:
    ...

while condition1 or condition2:
    ...
```

Rule 1: `C1 and C2` is `True` if both `C1` and `C2` are `True`
Rule 2: `C1 or C2` is `True` if one of `C1` or `C2` is `True`

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

0.21 Lists are objects for storing a sequence of things (objects)

So far, one variable has referred to one number (or string), but sometimes we naturally have a collection of numbers, say degrees $-20, -15, -10, -5, 0, \ldots, 40$
Simple solution: one variable for each value

```
C1 = -20
C2 = -15
C3 = -10
...
C13 = 40
```

Stupid and boring solution if we have many values!
Better: a set of values can be collected in a list

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

Now there is one variable, `C`, holding all the values

0.22 List operations 1: initialization and indexing

Initialize with square brackets and comma between the Python objects:

```
L1 = [-91, 'a string', 7.2, 0]
```

Elements are accessed via an index: `L1[3]` (index=3).
List indices start at `0`: `0`, `1`, `2`, ... `len(L1)-1`.

```
>>> mylist = [4, 6, -3.5]
>>> print(mylist[0])
4
>>> print(mylist[1])
6
>>> print(mylist[2])
-3.5
>>> len(mylist)  # length of list
3
```

0.23 List operations 2: append, extend, length

```
>>> C = [-10, -5, 0, 5, 10, 15, 20, 25, 30]
>>> C.append(35)   # add new element 35 at the end
>>> C
[-10, -5, 0, 5, 10, 15, 20, 25, 30, 35]
>>> C = C + [40, 45]     # extend C at the end
>>> len(C)               # length of list
12
```

0.24 The for loop is used for iterating over a list

A for loop iterates over elements in a list, and performs operations on each:

```
for element in list:
    <statement 1>
    <statement 2>
    ...
<first statement after loop>
```

- Simpler than the while loop (no conditional needed)
- Slightly less flexible

0.25 Example: For loop for temperature conversion

Task: Create a list of Celsius values similar to the previous one. Use a for-loop to iterate over the list, compute the corresponding Fahrenheit values and printing the values to the screen

0.26 Loop over elements in a list with a for loop

Use a *for* loop to loop over a list and process each element:

```
degrees = [0, 10, 20, 40, 100]
for C in degrees:
    print('Celsius degrees:', C)
    F = 9/5.*C + 32
    print('Fahrenheit:', F)
print('The degrees list has', len(degrees), 'elements')
```

As with *while* loops, the statements in the loop must be indented!

0.27 Making a table with a for loop

Table of Celsius and Fahreheit degrees:

```
Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15,
             20, 25, 30, 35, 40]
for C in Cdegrees:
    F = (9.0/5)*C + 32
    print(C, F)
```

Note: `print(C, F)` gives ugly output. Use an f-string to nicely format the two columns:

```
print(f'{C:5d} {F:5.1f}')
```

Output:

```
-20  -4.0
-15   5.0
-10  14.0
 -5  23.0
  0  32.0
  ......
 35  95.0
 40 104.0
```

0.28 A for loop can always be translated to a while loop

The for loop

```
for element in somelist:
    # process element
```

can always be transformed to a corresponding while loop

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

But not all while loops can be expressed as for loops!

0.29 Storing the table columns as lists

Let us put all the Fahrenheit values in a list as well:

```
Cdegrees = [-20, -15, -10, -5, 0, 5, 10,
            15, 20, 25, 30, 35, 40]
Fdegrees = []            # start with empty list
for C in Cdegrees:
    F = (9.0/5)*C + 32
    Fdegrees.append(F)   # add new element to Fdegrees
print(Fdegrees)
```

`print(Fdegrees)` results in

```
[-4.0, 5.0, 14.0, 23.0, 32.0, 41.0, 50.0, 59.0,
 68.0, 77.0, 86.0, 95.0, 104.0]
```

0.30 Using `range` to loop over indices

Sometimes we don't have a list, but want to repeat an operation N times. The Python function `range` returns a list of integers:

```
C = 0
for i in range(N):
    F = (9.0/5)*C + 32
    print(F)
```

- `range(start, stop, inc)` generates a list of integers `start`, `start+inc`, `start+2*inc`, and so on up to, *but not including*, `stop`.
- `range(stop)` is short for `range(0, stop, 1)`.

(In Python 3, range returns an *iterator*, which is not strictly a list, but behaves similarly when used in a `for` loop.)

0.31 Implement a mathematical sum via a loop

$$S = \sum_{i=1}^{N} i^2$$

```
N = 14

S = 0
for i in range(1, N+1):
    S += i**2
```

Or (less common):

```
S = 0
i = 1
while i <= N:
    S += i**2
    i += 1
```

Mathematical sums appear often so remember the implementation!

0.32 How can we change the elements in a list?

Say we want to add 2 to all numbers in a list:

```
>>> v = [-1, 1, 10]
>>> for e in v:
...     e = e + 2
...
>>> v
[-1, 1, 10]    # unaltered!!
```

0.33 Changing a list element requires assignment to an indexed element

What is the problem?

Inside the loop, `e` is an ordinary (`int`) variable, first time `e` becomes 1, next time `e` becomes 3, and then 12 - but the list `v` is unaltered

Solution: must *index a list element* to change its value:

```
>>> v[1] = 4    # assign 4 to 2nd element (index 1) in v
>>> v
[-1, 4, 10]
>>>
>>> for i in range(len(v)):
...     v[i] = v[i] + 2
...
>>> v
[1, 6, 12]
```

0.34 List comprehensions: compact creation of lists

Example: compute two lists in a for loop.

```
n = 16
Cdegrees = [];  Fdegrees = []  # empty lists

for i in range(n):
    Cdegrees.append(-5 + i*0.5)
    Fdegrees.append((9.0/5)*Cdegrees[i] + 32)
```

Python has a compact construct, called *list comprehension*, for generating lists from a for loop:

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

General form of a list comprehension:

```
somelist = [expression for element in somelist]
```

where `expression` involves `element`

0.35 Traversing multiple lists simultaneously with zip

Can we have one loop running over two lists? Solution 1: loop over indices

```
for i in range(len(Cdegrees)):
    print(Cdegrees[i], Fdegrees[i])
```

Solution 2: use the `zip` construct (more "Pythonic"):

```
for C, F in zip(Cdegrees, Fdegrees):
    print(C, F)
```

Example with three lists:

```
>>> l1 = [3, 6, 1];  l2 = [1.5, 1, 0];  l3 = [9.1, 3, 2]
>>> for e1, e2, e3 in zip(l1, l2, l3):
...     print(e1, e2, e3)
...
3 1.5 9.1
6 1 3
1 0 2
```

0.36 Nested lists: list of lists

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

```
Cdegrees = list(range(-20, 41, 5)) #range returns an iterator, convert to a list
Fdegrees = [(9.0/5)*C + 32 for C in Cdegrees]

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

print(table1[0])    # the Cdegrees list
print(table1[1])    # the Fdegrees list
print(table1[1][2])  # the 3rd element in Fdegrees
```

0.37 Extracting sublists (or slices)

We can easily grab parts of a list:

```
>>> A = [2, 3.5, 8, 10]
>>> A[2:]   # from index 2 to end of list
[8, 10]

>>> A[1:3]  # from index 1 up to, but not incl., index 3
[3.5, 8]

>>> A[:3]   # from start up to, but not incl., index 3
[2, 3.5, 8]

>>> A[1:-1] # from index 1 to next last element
[3.5, 8]

>>> A[:]    # the whole list
[2, 3.5, 8, 10]
```

Note: sublists (slices) are *copies* of the original list!

0.38 Iteration over general nested lists

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

Loops over list indices:

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

Loops over sublists:

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

0.39 Iteration over a specific nested list

```
L = [[9, 7], [-1, 5, 6]]
for row in L:
    for column in row:
        print(column)
```

Simulate this program by hand!

Question. How can we index element with value 5?

0.40 Tuples are constant lists

Tuples are constant lists that cannot be changed:

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

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

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

Tuples can do much of what lists can do:

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

0.41 Why tuples when lists have more functionality?

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

0.42 Key topics from this chapter

- While loops
- Boolean expressions
- For loops
- Lists
- Nested lists
- Tuples

0.43 Summary of loops, lists and tuples

While loops and for loops:

```
while condition:
    <block of statements>

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

Lists and tuples:

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

0.44 List operations self study: insert, delete

```
>>> C
[-10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
>>> C.insert(0, -15)     # insert -15 as index 0
>>> C
[-15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
>>> del C[2]             # delete 3rd element
>>> C
[-15, -10, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
>>> del C[2]             # delete what is now 3rd element
>>> C
[-15, -10, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

0.45 List operations self study: search, negative indices

```
>>> C.index(10)   # index of the first element with value 10
3
>>> 10 in C       # is 10 an element in C?
True
>>> C[-1]         # the last list element
45
>>> C[-2]         # the next last list element
40
>>> somelist = ['book.tex', 'book.log', 'book.pdf']
>>> texfile, logfile, pdf = somelist  # assign directly to variables
>>> texfile
'book.tex'
>>> logfile
'book.log'
>>> pdf
'book.pdf'
```

0.46 List operations self study: summary

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