Ch.2: Loops and lists

Joakim Sundnes[1,2] Hans Petter Langtangen[1,2]

Simula Research Laboratory[1]

University of Oslo, Dept. of Informatics[2]

Aug 29, 2018

Plan for 28 August

- Short quiz on topics from last week
- Exercise 1.4 and 1.12 from *Primer on Scientific Programming with Python*
- Loops and lists

Short quiz on last week's topic

We start with a quick quiz based on last week's lectures.

- The questions are supposed to test your understanding of last week's topic.
- You should try to answer these questions based on what you remember from last week, or by reasoning.
- Most of the questions become quite trivial if you test them in Python or try to google.

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
```

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).

Question 2

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

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

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
```

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)
```

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)
```

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)
```

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)
```

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?

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:
x0 = 14.0

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

Main topics of Chapter 2

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

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?

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)
# #if FORMAT == 'ipynb'
# ...
# #else
...
# #endif
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 misprint
- 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*

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

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.

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
```

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

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
```

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
```

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
# #if FORMAT == 'ipynb'
# ...
# #else
...
# #endif
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

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
```

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
```