Ch.3: Functions and branching

Ole Christian Lingjærde, Dept of Informatics, UiO

September 4-8, 2017

Today's agenda

- A small quiz to test understanding of lists
- Live-programming of exercises 2.7, 2.14, 2.15
- Introducing functions in programming

Before we start

Quizes like the one we are to start now will occur on later lectures as well.

Please note that:

- The questions are designed to test your *understanding*.
- Questions are (usually) trivial to solve by computer.
- Do *not* use your computer to solve them!

Quiz 1 (Warm up)

Suppose we perform the following program:

```
a = [1]

b = a + a

c = a[0] + a[0]

d = [1]*10

e = [2*i for i in range(10)]

f = [2*i for i in range(10) if i%3==0]
```

What are the values of the variables `b`, `c`, `d`, `e`, `f` ?

Answer to Quiz 1

```
a = [1]

b = a + a
# b = [1, 1]

c = a[0] + a[0]
# c = 2

d = [1]*10
# d = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

e = [2*i for i in range(10)]
# e = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

f = [2*i for i in range(10) if i%3==0]
# f = [0, 6, 12, 18]
```

Quiz 2 (List levels)

For new programmers, lists can be confusing. Make it a habit to always be aware of the number of levels in a list.

```
a = [1, 2, 3]          # One level
b = [[1,2], [3,4]]     # Two levels
c = [b]                # Three levels
d = [1, [1,2]]         # Mixed levels (one and two)
```

- Suggest an application for a one-level list!
- Suggest an application for a two-level list!
- Suggest an application for a mixed-level list!

Answer to Quiz 2

- Application of one-level list: to store daily measurements of temperature for a given time period.
- Application of a two-level list: to store daily measurements of many different weather variables in a given time period.
- Application of a mixed-level list: to store daily measurements of various variables in a given time period, as well as the name, location and other information about the weather station.

Quiz 3 (Counting levels)

Answer the following questions:

```
a = [0, 1]          # How many levels?

b = a + a           # How many levels has b?

a.append([4, 5]) # How many levels has a now?

a = [a] + a         # How many levels has a now?
```

Answer to Quiz 3

Answer the following questions:

```
a = [0,1]
# This list has one level

b = a + a
# [0,1,0,1]  (one level)

a.append([4, 5])
# [0,1,0,1,[4,5]]  (one and two levels)

a = [a] + a
# [[0,1,0,1,[4,5]],0,1,0,1,[4,5]] (one, two and three levels)
```

Quiz 4 (More about lists)

Suppose `a = [1,2]` and `b = [4,5,6]`.

- What happens here: `c = a.append(b)`
- What happens here: `c = a + b`
- What happens here: `c = zip(a,b)`
- Explain in words what happens here: `c = a.index(1)`
- Explain in words what happens here: `c = a.insert(2,3)`

Exercise 2.7

Generate equally spaced coordinates

We want to generate n+1 equally spaced x coordinates in $[a, b]$. Store the coordinates in a list.

- Start with an empty list, use a for loop and append each coordinate to the list. Hint: With n intervals, corresponding to n + 1 points in $[a, b]$, each interval has length $h = (b - a)/n$. The coordinates can then be generated by the formula $x_i = a + ih$, $i = 0, ..., n + 1$.
- Use a list comprehension as an alternative implementation.

Filename: coor.

Exercise 2.14

Explore Python documentation

Suppose you want to compute with the inverse sine function. How do you do that in a Python program?

Hint: The math module has an inverse sine function. Find the correct name of the function by looking up the module content in the online Python Standard Library7 document or use pydoc, see Sect. 2.6.3.

Filename: inverseSine.

Exercise 2.15

Index a nested list

We define the following nested list:

q = [['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h']]

a) Index this list to extract

- the letter a
- the list ['d', 'e', 'f']
- the last element h
- the d element.

and explain why q[-1][-2] has the value g.

b) We can visit all elements of ‘q using this nested for loop:

```
for i in q:
    for j in range(len(i)):
        print(i[j])
```

What type of objects are `i` and `j` ?

Filename: indexNestedList.

Functions in Python

Mathematical functions:

```
from math import sin
y = sin(x)                  # sin(.) is a function
```

Nonmathematical functions:

```
k = range(2, 10, 2)     # range(.) is a function
print(len(k))           # print(.) and len(.) are functions
```

Methods (functions used via the dot syntax):

```
a = [5, 10, 40, 45]
print(a.index(10))      # index(.) is a function
a.append(50)            # append(.) is a function
a.insert(2, 20)         # insert(.) is a function
```

Functions make life easier

- Functions in Python give *easy access* to already existing program code written by others (such as `sin(x)`). And there is plenty of such code in Python.
- Functions also give access to code written by ourselves - in previous projects or as part of the current project.
- To use the code in a function, we do not have to understand (or even see) the code.
- All we need to understand is what goes in and what comes out.

Functions let us delegate responsibilities

- Usually, writing a program to solve a problem involves solving many smaller tasks and putting it all together.
- Functions allow us to delegate some of these tasks so all WE have to worry about is putting all the results together.
- **Analogy:** in a car factory, they put together various parts often made elsewhere.

Summary of functions

- Function = a collection of statements we can execute wherever and whenever we want
- Function can take *input objects* (arguments) and produce output objects (returned results)
- Functions help to organize programs, make them more understandable, shorter, reusable, and easier to extend

Python function for implementing a mathematical function

The mathematical function

$$f(x) = x^3 + 3x^2 - x + 2$$

can be implemented in Python as follows:

```
def f(x):
    return x***3 + 3*x**2 - x + 2
```

- Functions start with `def`, then the name of the function, then a list of arguments (here `x`) - the *function header*
- Inside the function: statements - the *function body*
- Wherever we want, inside the function, we can "stop the function" and return as many values/variables we want

Calling a function

We distinguish between:

- **Defining** a function (we do this once in the program)
- **Calling** the function (can be done multiple times)

To call (use) it, we give its name and required arguments.

Example:

```
def f(x):                                  # Here f is defined
    return x***3 + 3*x**2 - x + 2

x = 1.5
print(f(x))                          # Calling f
y = f(x) + f(x/2)                    # Calling f twice
z = f(f(x)) + f(sin(x))              # Calling f three times
z = [f(x) for x in [0,1,2,3]]  # Calling f four times
```

A function can have multiple arguments

A Python function can have any number (0,1,2,...) of arguments.

Examples:

```
def day():              # Function with no arguments
    import time
    day = time.gmtime().tm_yday
    year = time.gmtime().tm_year
    return 'It is now the %gth day of %g' % (day,year)

def findMax(x,i,j): # Function with three arguments
    maxval = x[i]
    for k in range(i+1,j+1):
        if x[k] > maxval:
            maxval = x[k]
    return maxval

day()
# String returned: 'It is now the 249th day of 2017'

findMax([1,2,6,3,4,7,2,3], 3, 4)
# Value returned: 4
```

A function can have multiple return values

A Python function can have any number (0,1,2,...) of return values.

Example: the function

```
def f(x):
    return [x**2, x**4]
```

returns a *list* with two elements and can be called as

```
y = f(3.0)     # Now y is a list with two elements
y,z = f(3.0)   # Now y and z are float values
```

A function with no return values need no `return` statement.

Another way of returning multiple values

Another way of returning multiple values (without the use of lists) is to simply comma separate the return values.

Example: the function

```
def f(x):
    return x**2, x**4
```

returns a *tuple* with two elements and can be called as

```
y = f(3.0)     # Now y is a tuple with two elements
y,z = f(3.0)   # Now y and z are float values
```

Make sure you see the difference between this solution and the one on the previous slide! Here, `y` is a *tuple* and not a list. Tuples cannot be modified in the same way as lists.

Function calling with named arguments

Argument names can be given explicitly when we call a function. We can then provide arguments in any order we like.

Example: suppose we have defined the Python function

```
def f(x,y):
    return x**2 - 2*x*y + y**2
```

Then these four function calls give identical result:

```
z = f(3, 4)          # Unnamed arguments
z = f(3, y=4)        # One named argument
z = f(x=3, y=4)      # Two named arguments
z = f(y=4, x=3)      # Two named arguments
```

Local variables in functions

All variables defined *inside* a function are *local variables*. They only exist when the function is being executed and they are not visible outside the function.

Example:

```
g = 4                                   # Global variable

def y(t, v0):
    g = 9.81                            # Local variable
    return v0*t - 0.5*g*t**2

print('g = %3.1f' % g)
z = y(1,1)
print('g = %3.1f' % g)
```

Here *both* print statements print out ‘g = 4.0´. The variable g defined inside the function y(..) is not visible outside the function.

Function arguments are local variables

Arguments in a function definition are local variables and therefore only visible inside the function.

Example:

```
def g(s,t):
    c0 = 1.05
    res = (s+t+c0)**2
    return res

y = g(3.5, 5.0)
```

Calling a function

The call `g(3.0, 5.0)` leads to execution of these statements:

```
s = 3.5
t = 5.0
c0 = 1.05
res = (s+t+c0)**2
return res
```

Here, `s`, `t`, `c0`, `res` are local variables in the function.

Global variables are visible inside functions

Any variable defined *outside* the function can be accessed inside the function.

```
pi = 3.14

def area(r):
    res = pi * r * r
    return res

print('The area of a circle with radius %g is %g' % (2, area(2)))
```

In this case, `pi` is a global variable and is used inside the function.

Local variables can hide global variables

If a local variable and a global variable have the same name, only the local variable is visible inside the function.

Example:

```
def g(t):
    alpha = 1.0
    beta = 2.0
    return alpha + beta*t

print(g(1))          # Prints out '3.0'
alpha = 10.0
print(g(1))          # Still prints out '3.0'
```

In this example, the value `alpha = 10.0` is never actually used.

What is the purpose of hiding global variables?

- It can be very useful to access global variables inside a function, for example to access constants defined outside the function.
- Still, the rule is that when a name collision occurs, the local variable "wins" and the global variable becomes invisible
- Why? Because otherwise it would be impossible to know how a function would behave when used in new contexts (with new global variables).

Changing global variables in a function

Suppose we wanted to change the value of a global variable from inside a function. Not as easy as it seems:

```
x = 10
def f(y):
    x = 5    # We try to change the global variable
    return x + y

print(x)     # Prints out '10'
print(f(0)) # Prints out '5'
print(x)     # Prints out '10' (so the global variable x is still 10!)
```

Attempting to change a global variable inside a function fails in this case, because we inadvertently define a *local* variable `x` when we write `x=5`.

Changing global variables in a function (2)

- If we really want to change a global variable inside a function, we have to declare the variable as *global*.
- However, you should *only* do this if you really have to.

Example:

```
x = 10
def f(y):
    global x          # This says: don't create a local variable x
    x = 5             # This time we do change the global variable
    return x + y

print(x)     # Prints out '10'
print(f(0)) # Prints out '5'
print(x)     # Prints out '5' (so the global variable x is changed)
```

Example: Compute a function defined as a sum

This function approximates $\ln(1+x)$ for $x \geq 1$:

$$L(x,n) = \sum_{i=1}^{n} \frac{1}{i} \left(\frac{x}{1+x} \right)^i$$

Corresponding Python function:

```
def L(x,n):
    s = 0
    for i in range(1, n+1):
        s += (1.0/i)*(x/(1.0+x))**i
    return s
```

Example of use:

```
import math
x = 5.0
print (L(x, 10), L(x, 100), math.log(1+x))
```

Returning errors as well from the L(x, n) function

Suppose we want to return more information about the approximation:

- The first neglected term in the sum
- The error $(\ln(1+x) - L(x;n))$

```
def L2(x,n):
    s = 0
    for i in range(1, n+1):
        s += (1.0/i)*(x/(1.0+x))**i
    first_neglected_term = (1.0/(n+1))*(x/(1.0+x))**(n+1)
    import math
    exact_error = math.log(1+x) - s
    return s, first_neglected_term, exact_error

# typical call:
x = 1.2; n = 100
value, approximate_error, exact_error = L2(x, n)
```

Keyword arguments are useful to simplify function calls and help document the arguments

Functions can have arguments of the form `name=value`, and these are called *keyword arguments*.

Example:

```
def printAll(x, y, z=1, w=2.5):
    print(x, y, z, w)
```

Examples on calling functions with keyword arguments

```
>>> def somefunc(arg1, arg2, kwarg1=True, kwarg2=0):
>>>     print (arg1, arg2, kwarg1, kwarg2)

>>> somefunc('Hello', [1,2])    # drop kwarg1 and kwarg2
Hello [1, 2] True 0             # default values are used

>>> somefunc('Hello', [1,2], kwarg1='Hi')
Hello [1, 2] Hi 0               # kwarg2 has default value

>>> somefunc('Hello', [1,2], kwarg2='Hi')
Hello [1, 2] True Hi            # kwarg1 has default value

>>> somefunc('Hello', [1,2], kwarg2='Hi', kwarg1=6)
Hello [1, 2] 6 Hi               # specify all args
```

If we use `name=value` for *all* arguments *in the call*, their sequence can in fact be arbitrary:

```
>>> somefunc(kwarg2='Hello', arg1='Hi', kwarg1=6, arg2=[2])
Hi [2] 6 Hello
```

How to implement a mathematical function of one variable, but with additional parameteres?

Consider a function of t, with parameters A, a, and ω:

$$f(t; A, a, \omega) = Ae^{-at}\sin(\omega t)$$

Possible implementation

Python function with t as positional argument, and A, a, and ω as keyword arguments:

```
from math import pi, exp, sin

def f(t, A=1, a=1, omega=2*pi):
    return A*exp(-a*t)*sin(omega*t)

v1 = f(0.2)
v2 = f(0.2, omega=1)
v2 = f(0.2, 1, 3)  # same as f(0.2, A=1, a=3)
v3 = f(0.2, omega=1, A=2.5)
v4 = f(A=5, a=0.1, omega=1, t=1.3)
v5 = f(t=0.2, A=9)
v6 = f(t=0.2, 9)   # illegal: keyword arg before positional
```