Exercise 1

Try the following code segment:

```
>>> list = [3,4,2,1]
>>> for number in list:
        print 'item %s in list %s' % (number, list)
        if number > 2:
            list.remove(number)
```

Pay attention to the print line

- The "placeholder" %s says that in that position you have to enter a **s**tring, which is taken from the variable that follows the second %, outside the first string. You can use more than one placeholder, one for each variable needed.
- Some common print formats:
  1. %d = integer
  2. %e = float in scientific notation
  3. %11.3e = float with 3 decimals, field of 11 characters
  4. %.3f = float in fixed decimal notation, minimum length 3 decimals
  5. %s = string
  6. %-20s = string with a field of 20 characters, left aligned

Exercise 1

```
>>> list = [3,4,2,1]
>>> for number in list:
        print 'item %s in list %s' % (number, list)
        if number > 2:
            list.remove(number)
```

Now let's analyze better the code:

- What do you think it should do in the programmer's intentions ?
- After the loop...what list do you obtain, and why?
- The problem is that in the second iteration of the loop, the for loop uses the index 1, e then skips the second list item, the "4", that now has index 0
- The message is simple:
  never ever change a list (or any mutable object) on which you are iterating!
- How can you get what you really wanted? With a copy!
  - *Write the correct code that removes every item > 2 with a for loop.*
  - *Write the code that removes all the items > 2 from the original list, but use a while loop and an index that is properly updated on each cycle*

Exercise 2

Still about the subject of the previous problem, does the following code clear the list?

```
>>> list = ['a','b','c','d']
>>> for item in list:
        del list[0]
```

Try to understand what happens, printing for debug the list and the current item in the loop

- *Write a while loop that using the statement*

  `del list[0]`

  is able to correctly delete the list

Istituto Nazionale di Fisica Nucleare

Solutions 1

- ```
  list=[3,4,2,1]
  for item in list[:]:
      if item > 2:
          list.remove(item)
  ```

- ```
  list=[3,4,2,1]
  maxIndex=len(list)-1
  index=0
  while index <= maxIndex:
      if list[index]>2:
          list.remove(list[index])
          maxIndex=maxIndex-1     #the list is smaller
      else:
          index=index+1           #the index is updated
  print list                      #only if the list size
                                  #is not changed
  ```

Solutions 2

```
list=['a','b','c','d']
i=len(list)-1          #the index starts from 0
while i>=0:
   del list[0]
   i=i-1               #if an item is removed
print list             #the loop index i is
                       #decremented
```

Istituto Nazionale di Fisica Nucleare

Scope

- ***Variable scope*** = part of the code in which you can use the given variable
- There are three different *scopes*, that allow to identify the variables within functions:

1. If a variable is assigned inside a *def,* it is *local* to that function.
2. If a variable is assigned outside any *def,* it is *global* to the entire code.
3. If a variable is assigned inside a *def*, that includes a second *def*, it is *nonlocal* for the nested function (3.x)

```
def outer():
    x = 1
    def inner():
        nonlocal x
        x += 1
        print(x)
    return inner
```

- `global` is the keyword for the global scope, in Python 3.x `nonlocal` is the keyword for the external nearest scope, that is not necessarily the global one.
- Please AVOID to modify global variables inside a function!

LEGB rule

Built-in (Python):
preassigned names in standard modules: open, range, Syntax Error

Global (modul):
names assigned to the top-level of a module file, or declared as global in a def within the file

Enclosing function locals:
names in the local scope of each function that includes the starting function

Local(function):
names assigned within a function, and not declared global

Python looks for the variables in this order: first the local scope, then the scope of the functions that include the starting function, then the global scope and finally the built-in.

The first name occurrence WINS, and hides the subsequent ones.
Usually the place in which is placed a variable determines its scope.

Exercise 3

What do these codes print, and why?

```
>>> x = "prisma"
>>> def func():
        print x
>>> func()
```

```
prisma
```

Since the variable x is not assigned inside the function, it is considered global (you can use the global variables inside a function)

```
>>> x = "prisma"
>>> def func():
        x='Master'
        print x
>>> func()
>>> print x
```

```
Master
prisma
```

The assignment of x inside the function hides the global x, so the x='Master' is local to the function and does not modify the global x

```
>>> x = "prisma"
>>> def func():
        global x
        x='Master'
>>> func()
>>> print x
```

```
Master
```

The global statement force the assignment in the function to refer to the variable x in the global scope, that now can be modified.

Lambda functions

- The functions are Python objects to all the effects, and can be used like any other type of data.
- You can pass a function as an argument to another function, or return a function as a return value from a function, or you can assign it to a variable, or include it in a list

```
>>> def myfunc(x):
        return x*3
>>> def applier(q,x)
        return q(x)
>>> applier(myfunc,7)
21
```

- You can define a function without giving it an explicit name. It can be useful in examples like the previous one, where you have to pass a "small" function as an argument to another function:

```
>>> applier(lambda z: z*3, 7)
21
```

Lambda functions

- `lambda <args>: <expression>`

  is equivalent to a function with arguments `<args>` and `<expression>` as the return value.

- Note that only functions that have a SINGLE expression can be defined by this notation

Istituto Nazionale di Fisica Nucleare

filter map reduce

These are three functions that are useful, when used in conjuction with lambda functions and lists, to build new lists starting from existing ones.

- **`filter(function, iterable)`**
  builds a list formed from the items of *iterable* for which the function *function* returns *True(1)*

- **`map(function, iterable, ...)`**
  builds a list formed from the values `function(item)` computed for each `item` of `iterable`.

- **`reduce(function, iterable)`**
  returns a single value computed applying the function on the first two elements of the sequence, then on the pair formed by the first result and the third item, and so on…

filter

This is the "Sieve of Eratostene", that finds the prime numbers from 2 to n included:

```
def eratostene(n):
    setaccio = range(2, n+1)
    primi = []
    while setaccio:
        primi.append(setaccio[0])
        setaccio = filter(lambda x: x%primi[-1], setaccio)
    return primi
```

- It creates a list of all natural numbers from 2 to n, then adds the first number of the sieve to the list of prime numbers found, and removes from the sieve all its multiples. Then it continues until all the numbers in the sieve are removed.

- `%` is the modul operator (returns the remainder). Here the filter function builds a new sieve and deletes multiples of the considered prime number (the last one in the list `primi`) in a single instruction.

map reduce

- We want to calculate the longest string in a sequence:

```
>>> values=('abc','xy','abcdef','xyz')
>>> mapped=map(lambda  x:len(x),values)
>>> max_len=reduce(lambda  x,y: max(x,y),mapped)
```

- The most interesting part of the matter: if the functions are associative and commutative, as in this case, the problem is easily parallelizable.

- In principle you could send each map operation to a different computing core, and even the reduce may be performed in multiple parallel steps.

- From Python 2.6 you can use the multiprocessing.Pool.map () function:

```
>>> import operator
>>> from multiprocessing import Pool
>>> strings = ['string 1', 'string xyz', 'foobar']
>>> pool = Pool(processes=2)
>>> print reduce(operator.add, pool.map(len, strings))
24
```

Variable number of arguments

In Python you can define functions with a variable number of arguments

```
def somefunc(a, b, *args):
    # args is a tuple of all supplied positional arguments
    ...
    for arg in args:
        <work with arg>
```

- A double asterisk indicates a variable length set of **named** arguments (a dictionary), which must be passed to the function when it is called.

```
def somefunc(a, b, *args, **kwargs):
# args is a tuple of all supplied positional arguments
# kwargs is a dictionary of all supplied keyword arguments
...
for arg in args:
    print arg
for key in kwargs.keys():
    print key, ' : ', kwargs[key]
```

Example

Write a function that compute the average, minimum and maximum of all input arguments. It has as a input a variable number of arguments (must be numbers), and returns as output the tuple `(average, min, max)`.

```
def statistics(*args):
    avg = 0; n = 0;         # avg and n are local variables
    for number in args:     # sum up all numbers (arguments)
        n += 1
        avg += number
    avg /= float(n)
    min = args[0]; max = args[0]
    for term in args:
        if term < min: min = term
        if term > max: max = term
    return avg, min, max
>>> average, vmin, vmax = statistics(v1, v2, v3, b)
>>> print 'average =', average, 'min =', vmin, 'max=', vmax
```

http://bit.do/statistics-py

We are returning local variables ??? No, we're returning a reference to an object in memory, which will therefore continue to exist even when the function is finished.
By the way, the function can be MUCH simpler:

```
def statistics(*args):
    return sum(args)/float(len(args)), min(args), max(args)
```

```
def table_things(**kwargs):
    for name, value in kwargs.items():
        print '%s = %s' % (name, value)


>>> table_things(apple = 'fruit', cabbage = 'vegetable')
cabbage = vegetable
apple = fruit
```

Istituto Nazionale di Fisica Nucleare

Scripts

Script

- You can turn what is written in IDLE, or directly in the Python interpreter shell, into an executable program

- On Windows you can use
  http://www.py2exe.org/
  an extension that allows you to get executables that do not require a prior installation of Python on the target computer.

- On Linux an executable Python program is a *script*, a textual code.

Hello World!

Maybe it's a bit late, in the slide 84, to introduce the classic example Hello World… but I need it to show you some characteristics of the scripts.
Our script, however, is deliberately a bit more complicated than usual.

http://bit.do/HelloWorld-py

```
#!/usr/bin/env  python

import  sys,math
usage  =  'Usage:  %s number'  %  sys.argv[0]
try:
    number=sys.argv[1]
except:
    print usage
    sys.exit(1)

r=float(number)
s=math.sin(r)
print "Hello,  World!  sin("  +  str(r)  +  ")="  +  str(s)
```

```
#!/usr/bin/env python

import sys,math
usage = 'Usage: %s number' % sys.argv[0]
try:
    number=sys.argv[1]
except:
    print usage
    sys.exit(1)

r=float(number)
s=math.sin(r)
print "Hello, World! sin(" + str(r) + ")=" + str(s)
```

- In the first row it should appear the full path to the interpreter of the script, but you can avoid to explicitly write the path leaving to the **env** program, which should be more standard, to find python in the system PATH.
- This makes it possible to move the script between multiple systems with python installations homed in different directories
- As usual to run the script you have to change its permissions

```
#!/usr/bin/env python

import sys,math
usage = 'Usage: %s number' % sys.argv[0]
try:
    number=sys.argv[1]
except:
    print usage
    sys.exit(1)

r=float(number)
s=math.sin(r)
print "Hello, World! sin(" + str(r) + ")=" + str(s)
```

```
number = sys.argv[1]
```

- The first item of the list `sys.argv[]` contains the script name (remember: the starting index is ALWAYS `0`), the other items contain the script arguments, passed from the command line.

```
r = float(number)
```

- You can not make a calculation directly with the string, so you need to convert it to a floating-point number.

  Remember that the type of Python variables are dynamically assigned, and the type is strong: you cannot multiply a string and a float number without an explicit casting.

```
s = math.sin(r)
```

- At this point the function that calculates the `sin(r)` is called, and that function is defined inside the math module. Note that the variable s, as well as the other variables used, has not been declared before using it, and it will contain the floating point calculation result.

```
#!/usr/bin/env python

import sys,math
usage = 'Usage: %s number' % sys.argv[0]
try:
    number=sys.argv[1]
except:
    print usage
    sys.exit(1)

r=float(number)
s=math.sin(r)
print "Hello, World! sin(" + str(r) + ")=" + str(s)
```

```
try:
    <code that might cause a runtime error>
except:
    <error recovering code>
```

The keyword **try** identifies a block of code to “try", and that can be the source of a runtime error.

The keyword **except** identifies a block of code that is executed if the try block raises an error

As usual the code must be indented below the keywords.

```
#!/usr/bin/env python

import sys,math
usage = 'Usage: %s number' % sys.argv[0]
try:
    number=sys.argv[1]
except:
    print usage
    sys.exit(1)

r=float(number)
s=math.sin(r)
print "Hello, World! sin(" + str(r) + ")=" + str(s)
```

- A module is simply a text file containing the Python code, and ending with the extension .py
- How can you use a module in a script? You can use the keyword **import** followed by the name of the file without the extension py.
- The import command includes in the script namespace, from that moment on, all the functions defined within the module.
- If you want to add your own module's directory to Python use:

```
>>> module_dir=os.path.join(os.environ['HOME'],'my','modules')
>>> sys.path.insert(0,module_dir)
```

Built-in modules and extensions

- The standard Python distribution already has many modules available, and many others can be found with internet: The Python Package Index (`PyPI`) for example is a place where you can find other modules, and you can also publish your custom module .

  https://pypi.python.org/pypi

- To install the "manager" of the repository, you need to install the package `python-pip`.

- When you have find the right module, you can install it with:

```
root@SL64 ~# pip install <package name>
```

Custom modules

- To be able to transform your function in a module, you have no choice but to save it in a file with an appropriate name.
- Let's go back to our example, to the recursive function we wrote that prints the items of nested lists. Save the function in the file `nester.py` in your home directory.

```
>>> def print_lol(the_list):
        for each_item in the_list:
            if isinstance(each_item, list):
                print_lol(each_item)
            else:
                print each_item
```

- Now try to import the module, and use the function `print_lol(test_list)`. What happens?

```
>>> import nester
>>> test_list=['1','2','3']
>>> print_lol(test_list)
```

Namespaces

- The problem is that every Python code has an associated ***namespace***. In the IDLE shell, and generally in the main program, the active namespace is

  `__main__`

  When you create a new module, a new namespace with the same module name is automatically created, and the functions defined in the module are available in that namespace.

- The function `print_lol` defined in the `nester` module can be used, after the `import` command seen above, only making an explicit reference to the right *namespace*:

```
>>> import nester
>>> test_list=['1','2','3']
>>> nester.print_lol(test_list)
```

import

1. **import *moduleName***

   Any function (or class) defined in the module file `moduleName.py` is imported and useable in the script, but you must prepend moduleName to the name of the function

   ```
   moduleName.myFunction (34)
   ```

2. **from *moduleName* import ***

   Any function (or class) defined in the module file `moduleName.py` is imported and useable in the script with its name:

   ```
   myFunction(34)
   ```

3. **from *moduleName* import myFunction**

   Only the myFunction is imported from the moduleName, directly in the main namespace:

   ```
   myFunction(34)
   ```

What is the best method?

- There isn’t a best method.
- Be warned that the methods 2 and 3 make the main namespace "dirty", with the inclusion of new functions in it.
  If in the module is defined a function with a name already used by another function defined in the `__main__` namespace, this last function is "overwritten"!
- If you write the module with IDLE, and save and run it with F5, the module functions are imported in the main namespace and you can easily debug them.

Documentation

- Back to our file `nester.py`, **surely** you have realized that something is missing: the documentation.

- In Python, a standard technique to write the documentation of functions and modules is to use a string delimited by **three** consecutive quotes, double or single: if the string is not assigned to a variable, everything that is bordered by triple quotes is a comment.

- All the lines that begin with # are also comments; this can be very useful for example to temporarily disable a line of code