Programming 1

Laszlo SZATHMARY

University of Debrecen
Faculty of Informatics

Lecture #8
- modules
- random numbers

(last update: 2025-01-18 [yyyy-mm-dd])

2024-2025, 2nd semester

Modules

As our program gets longer, the need arises to cut it into several source files.

First, maintenance would be easier. The project would be clearer, simpler.

Second, we might want to reuse some useful functions in different programs, without copying the functions to every program.

Exercise:

Write two programs:

1. Print prime numbers that are less than 100 (`ex1.py`)
2. Print the sum of prime numbers that are less than 200 (`ex2.py`)

Link: https://arato.inf.unideb.hu/szathmary.laszlo/pmwiki/index.php?n=EnPy3.20121110a

ex1.py

```
def is_prime(n):
    …
```

ex2.py

```
def is_prime(n):
    …
```

When ready, let's investigate what is common in the two scripts. The common part is the `is_prime()` function.

1. the very same function is repeated at different places
2. the `is_prime()` function performs an operation that could be useful in the future in another project

pylab.py

```
def is_prime(n):
    …
```

ex1.py

```
import pylab
…
```

ex2.py

```
import pylab
…
```
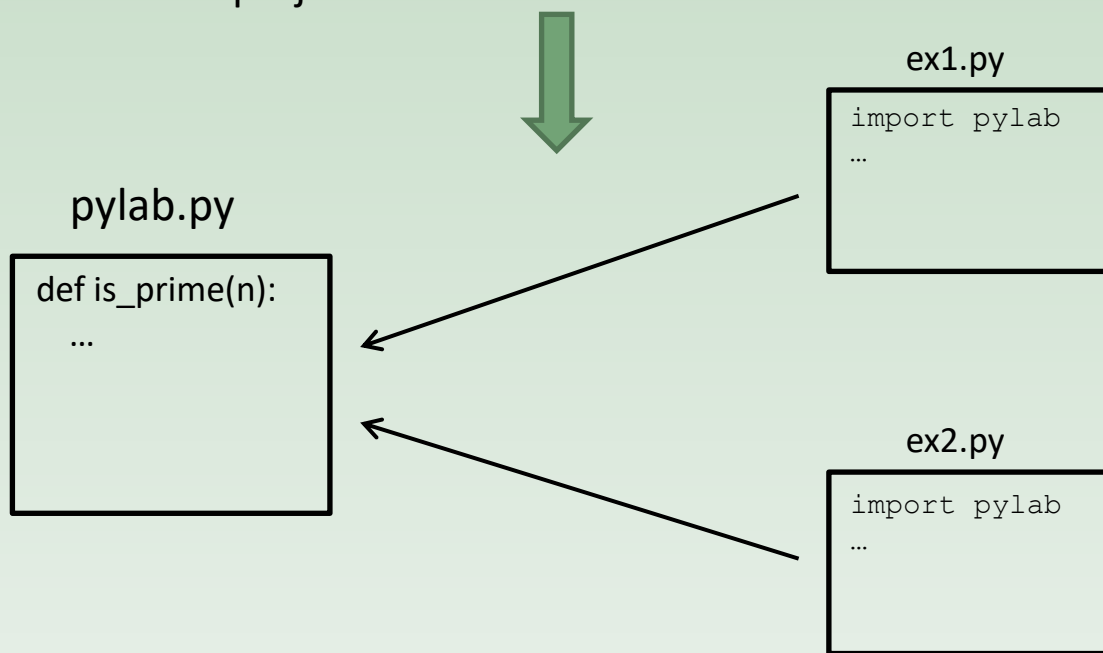
Move the common function to a module (e.g. `pylab.py`), and import this module in the scripts.

pylab.py

```python
 1  def is_prime(n):
 2      """
 3      Decide whether a number is prime or not.
 4      """
 5      if n < 2:
 6          return False
 7      if n == 2:
 8          return True
 9      if n % 2 == 0:
10          return False
11
12      i = 3
13      maxi = n**0.5 + 1
14      while i <= maxi:
15          if n % i == 0:
16              return False
17          i += 2
18
19      return True
20
21
22  def hello():
23      return "Hello, World!"
```

Usage (ex3.py):

```python
 8  import pylab
 9
10
11  def main():
12      print(pylab.is_prime(5))
13
14      print(pylab.hello())
```

```
import pylab       # without '.py'
```

Variations

`import pylab`

It won't put the functions defined in `pylab` to the current symbol table. Only the name of the module (`pylab`) is put in the symbol table.

If you want to use the module's functions, then refer to them using the module's name:

`pylab.is_prime(n)`

`import pylab as pl`

If the module has a long name and/or we want to refer to it several times, then you can rename it by putting an alias on it. Then:

`pl.is_prime(n)`

Variations (cont.)

```
from pylab import is_prime
```

Meaning: from the `pylab` module we import just the function name `is_prime` to the symbol table.
!!! It doesn't bring in the module's name to the symbol table **!!!**
Ex.: `print(pylab.hello())`   # error, the symbol „pylab" is unknown

Solution:

```
1. from pylab import is_prime, hello
```

```
2. from pylab import is_prime
   import pylab
```

```
3. from pylab import *
```
Not recommended, since we won't know what is imported from where. It just causes a mass.

Testing a module

Before:

```python
 1  def is_prime(n):
 2      """
 3      Decide whether a number
 4      is prime or not.
 5      """
 6      if n < 2:
 7          return False
 8      if n == 2:
 9          return True
10      if n % 2 == 0:
11          return False
12
13      i = 3
14      maxi = n**0.5 + 1
15      while i <= maxi:
16          if n % i == 0:
17              return False
18          i += 2
19
20      return True
21
22
23  def hello():
24      """
25      Classic example.
26      """
27      return "Hello, World!"
```

After:

```python
 1  #!/usr/bin/env python3
 2
 3  """
 4  pylab module
 5  ============
 6
 7  This module was made for the Python lab.
 8  Functions used often are collected here.
 9  """
10
11  def is_prime(n):
12      """
13      Decide whether a number is prime or not.
14      """
15      if n < 2:
16          return False
17      if n == 2:
18          return True
19      if n % 2 == 0:
20          return False
21
22      i = 3
23      maxi = n**0.5 + 1
24      while i <= maxi:
25          if n % i == 0:
26              return False
27          i += 2
28
29      return True
30
31
32  def hello():
33      """
34      Classic example.
35      """
36      return "Hello, World!"
37
38  ##############################
39
40  if __name__ == "__main__":
41      for n in range(2, 20):
42          if is_prime(n):
43              print("{n} is prime".format(n=n))
```

docstring

test

7

Testing a module (cont.)

```
37
38    ############################
39
40    if __name__ == "__main__":
41        for n in range(2, 20):
42            if is_prime(n):
43                print("{n} is prime".format(n=n))
```

arbitrary tests

The condition (line 40) is <u>true</u>, if the module is <u>executed directly</u> from the command-line (i.e. `./pylab.py`). In this case the test will run.

If the module is <u>imported</u>, then the condition is <u>false</u>, thus the test won't run.

That is: if we write a module, then (1) it can be used as a module, and (2) it can also be used as a stand-alone script!

Exercise: Modify the `pylab.py` module the followinf way. If it's executed directly, then it should read a number from the standard input, and then it should print whether this number is a prime or not.
Then execute the `ex1.py` script, in which import this modified `pylab.py` module. What do you notice?

Testing a module (cont.)

pylab.py

```
1   #!/usr/bin/env python3
2
3
4   def hello():
5       return "Hello, World!"
6
7   ###########################
8
9   if __name__ == "__main__":
10      print(__name__)
```

ex1.py

```
1   #!/usr/bin/env python3
2
3   import pylab
4
5
6   def main():
7       print(pylab.hello())
8       print(pylab.__name__)
9
10  ###########################
11
12  if __name__ == "__main__":
13      main()
```

```
$ ./pylab.py
__main__
$
$ ./ex1.py
Hello, World!
pylab
```

Testing a module (cont.)

```
37
38    ##############################
39
40    if __name__ == "__main__":
41        for n in range(2, 20):
42            if is_prime(n):
43                print("{n} is prime".format(n=n))
```

Another advantage:

If you add tests to the end of your module, then you provide some concrete examples about the usage of your module. If somebody wants to use your module, (s)he will get some examples in the form of your tests.
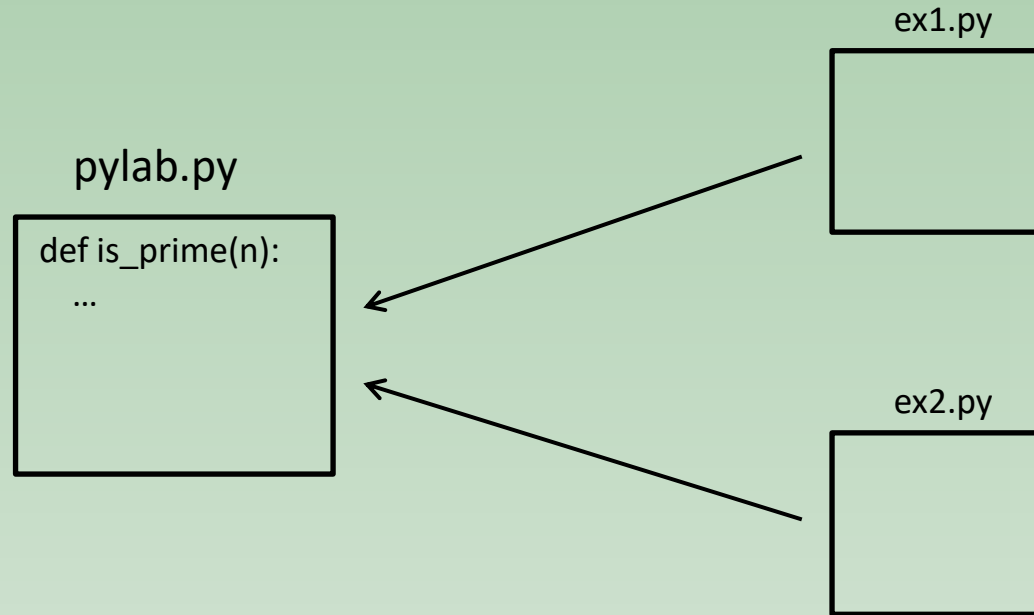
„Alice: - I wrote a cool module that I just sent to you in email. Did you
               get it?
Bob: - Yeah, but… How can I use these functions?
Alice: - Go down to the bottom. There you will see some examples.
Bob: - Oh, OK. Cool! Thanks!"

Modules

ex1.py

pylab.py

```
def is_prime(n):
    …
```

ex2.py

Another advantage of using modules

If you modify a function, then you need to do it at one place only.

> **Exercise:** In the `pylab.py` module replace the implementation of the `is_prime()` function to the much more efficient Miller-Rabin test.
> You will find the source code of the MR test in the course material.
>
> Then run the scripts `ex1.py` and `ex2.py` . What do you notice?

How a module is imported

```
import spam
```

We want to import this module. How will the interpreter find it? From where will it be imported?

1) The unterpreter checks if it's a built-in module.
   Built-in module: compiled in the interpreter.

```
>>> import sys
>>> sys.builtin_module_names
```

2) If it's not found, then it'll look for a file called `spam.py` in a list of directories. This list of directories can be found in `sys.path` .

```
>>> import sys
>>> sys.path
```

This `sys.path` is initialized the following way:
   - the folder containing the script
   - The `PYTHONPATH` environment variable. It is similar to `PATH` : a colon-separated list of directories.
   - system folders specific to your Python installation

How a module is imported (cont.)

```
>>> import sys
>>> sys.path
```

`sys.path` is a normal list => after the initialization, you can also modify it in your script

!!! The folder containing the current script will be at the *beginning* of `sys.path`, i.e. it comes *before* the folders of the standard library. It can cause some strange errors if the folder contains some files whose names are identical to the names of some modules in the standard library **!!!**

Random numbers

```
1   >>> import random
2   >>>
3   >>> random.random()
4   0.12214691572646652
5   >>> random.randint(1, 10)
6   9
7   >>> li = [3, 8, 2, 8, 4, 2, 1, 9]
8   >>> li
9   [3, 8, 2, 8, 4, 2, 1, 9]
18  >>> random.shuffle(li)
19  >>> li
20  [9, 8, 1, 4, 8, 2, 3, 2]
21  >>>
22  >>> random.choice(li)
23  8
```

[0.0, 1.0)

lower <= N <= upper

mixes (shuffles) the elements *in place* (random permutation)

selecting a random element from the list

Exercise: the `shuffle` method modifies the list *in place*. Write a function called `shuffled` that returns the shuffled list, allowing us to do the following for instance: `n = shuffled(li)[-1]`

14

Exercises

1. [20121110a] modules
2. [20120905c] shuffled
3. Implement the **queue** and **stack** data structures with classes. Document everything with docstrings: the module, the class, and the class' functions.