Public – Private key encryption using OpenSSL

Using OpenSSL on the command line you'd first need to generate a public and private key, you should password protect this file using the -passout argument, there are many different forms that this argument can take so consult the OpenSSL documentation about that.

```
$ openssl genrsa -out private.pem 1024
```

This creates a key file called private.pem that uses 1024 bits. This file actually have both the private and public keys, so you should extract the public one from this file:

```
$ openssl rsa -in private.pem -out public.pem -outform PEM -pubout
```

You'll now have public.pem containing just your public key, you can freely share this with 3rd parties.

You can test it all by just encrypting something yourself using your public key and then decrypting using your private key, first we need a bit of data to encrypt:

```
$ echo 'too many secrets' > file.txt
```

You now have some data in file.txt, lets encrypt it using OpenSSL and the public key:

```
$ openssl rsautl -encrypt -inkey public.pem -pubin -in file.txt -out file.ssl
```

This creates an encrypted version of file.txt calling it file.ssl, if you look at this file it's binary, nothing very useful to anyone. Now you can unencrypt it using the private key:

```
$ openssl rsautl -decrypt -inkey private.pem -in file.ssl -out decrypted.txt
```

You will now have an unencrypted file in decrypted.txt:

```
$ cat decrypted.txt<br>
too many secrets
```

Security

You need to use a Crypto library for python.

Python-RSA is a pure-Python RSA implementation. It supports encryption and decryption, signing and verifying signatures, and key generation according to PKCS#1 version 1.5: https://stuvel.eu/rsa >>> https://github.com/sybrenstuvel/python-rsa/

You can then install Python-RSA using the pip tool: `pip install rsa`

Or download it at `https://pypi.python.org/pypi/rsa`

Also https://www.dlitz.net/software/pycrypto/api/2.6/ / `https://pypi.python.org/pypi/pycrypto` contains python crypto libraries.

A note on Using Modules

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a *script*. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a *module*; definitions from a module can be *imported* into other modules or into the *main* module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended. Within a module, the module's name (as a string) is available as the value of the global variable__name__. For instance, use your favorite text editor to create a file called fibo.py in the current directory with the following contents:

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()
```

```
def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Now enter the Python interpreter and import this module with the following command:

```
>>>
```

```
>>> import fibo
```

This does not enter the names of the functions defined in fibo directly in the current symbol table; it only enters the module name fibo there. Using the module name you can access the functions:

```
>>>
```

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

If you intend to use a function often you can assign it to a local name:

```
>>>
```

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the *first* time the module name is encountered in an import statement. (They are also run if the file is executed as a script.)

Each module has its own private symbol table, which is used as the global symbol table by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand, if you know what you are doing you can touch a module’s global variables with the same notation used to refer to its functions, modname.itemname.

Modules can import other modules. It is customary but not required to place all import statements at the beginning of a module (or script, for that matter). The imported module names are placed in the importing module’s global symbol table.

There is a variant of the import statement that imports names from a module directly into the importing module’s symbol table. For example:

```
>>>
```

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken in the local symbol table (so in the example, fibo is not defined).

There is even a variant to import all names that a module defines:

```
>>>
```

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore (_). In most cases Python programmers do not use this facility since it introduces an unknown set of names into the interpreter, possibly hiding some things you have already defined.

Note that in general the practice of importing * from a module or package is frowned upon, since it often causes poorly readable code. However, it is okay to use it to save typing in interactive sessions.

Locating Modules

When you import a module, the Python interpreter searches for the module in the following sequences −

- The current directory.

- If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.
- If all else fails, Python checks the default path. On UNIX, this default path is normally /usr/local/lib/python/.

The module search path is stored in the system module sys as the sys.pathvariable. The sys.path variable contains the current directory, PYTHONPATH, and the installation-dependent default.

The PYTHONPATH Variable:

The PYTHONPATH is an environment variable, consisting of a list of directories. The syntax of PYTHONPATH is the same as that of the shell variable PATH.

Here is a typical PYTHONPATH from a Windows system:

```
set PYTHONPATH=c:\python20\lib;
```

And here is a typical PYTHONPATH from a UNIX system:

```
set PYTHONPATH=/usr/local/lib/python
```

Encryption and decryption using Python

1. Bob generates a keypair, and gives the public key to Alice. This is done such that Alice knows for sure that the key is really Bob's (for example by handing over a USB stick that contains the key).

Use :py:meth:`rsa.PrivateKey.load_pkcs1` and :py:meth:`rsa.PublicKey.load_pkcs1` to load the openSSL keys from a file:

```
>>> import rsa
>>> with open('private.pem', mode='rb') as privatefile:
...     keydata = privatefile.read()
>>> bob_priv = rsa.PrivateKey.load_pkcs1(keydata)
```

2. Alice writes a message, and encodes it in UTF-8. The RSA module only operates on bytes, and not on strings, so this step is necessary.

```
>>> message = 'hello Bob!'.encode('utf8')
```

3. Alice encrypts the message using Bob's public key, and sends the encrypted message.

```
>>> import rsa
>>> crypto = rsa.encrypt(message, bob_pub)
```

4. Bob receives the message, and decrypts it with his private key.

```
>>> message = rsa.decrypt(crypto, bob_priv)
>>> print(message.decode('utf8'))
hello Bob!
```

Since Bob kept his private key private, Alice can be sure that he is the only one who can read the message. Bob does not know for sure that it was Alice that sent the message, since she didn't sign it.