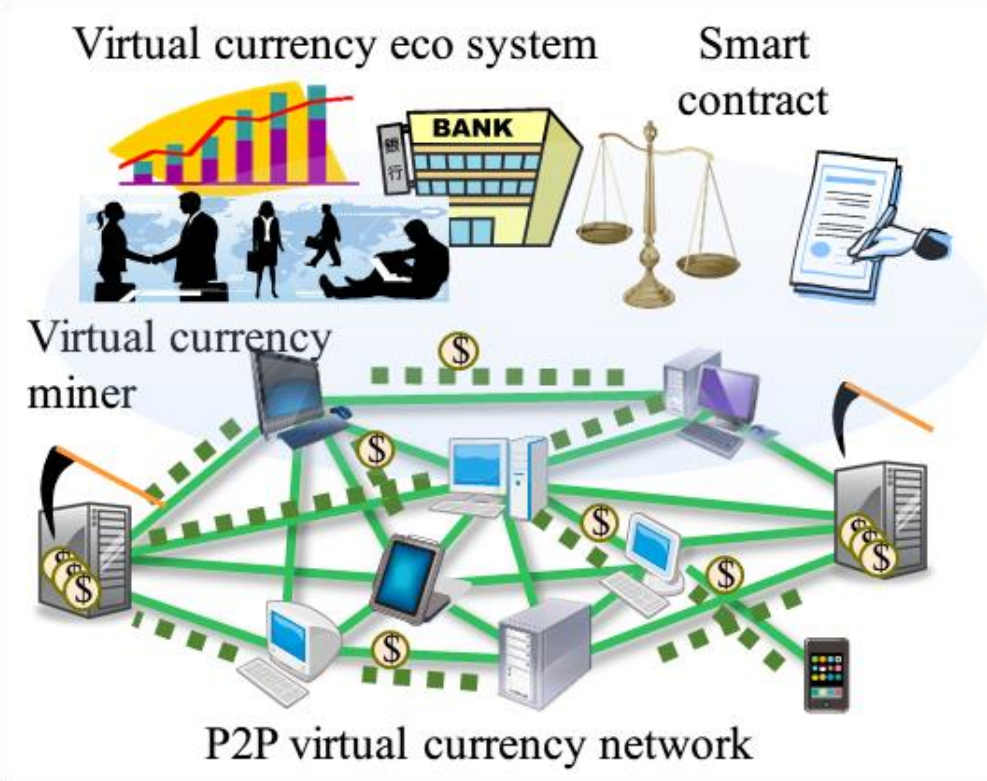
Modular Design

Ultra-large scale systems



Modules

The term **“module”** refers to the design and/or implementation of specific functionality to be incorporated into a program.

- SOFTWARE DESIGN
  - provides a means for the development of well-designed programs
- SOFTWARE DEVELOPMENT
  - provides a natural means of dividing up programming tasks
  - provides a means for the reuse of program code
- SOFTWARE TESTING
  - provides a means of separately testing parts of a program
  - provides a means of integrating parts of a program during testing
- SOFTWARE MODIFICATION AND MAINTENANCE
  - facilitates the modification of specific program functionalities

Module specification

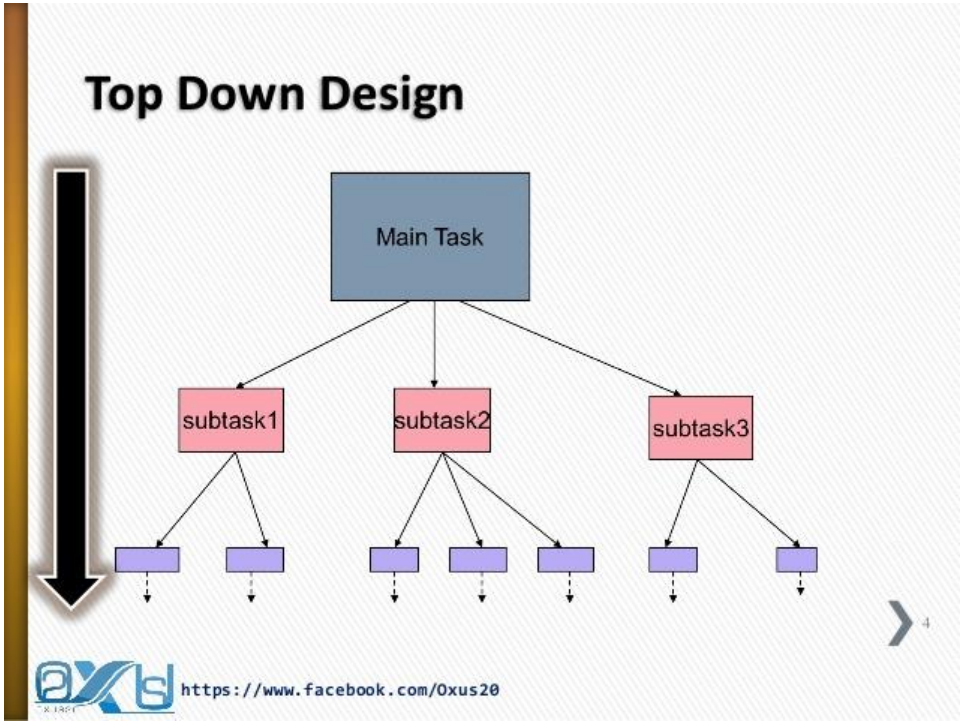
A module's interface is a specification of what it provides and how it is to be used. Any program code making use of a given module is called a **client** of the module. A **docstring** is a string literal denoted by triple quotes used in Python for providing the specification of certain program elements.

```
def numprimes(start, end):

    """ Returns the number of primes between start and end, inclusive.

        Returns -1 if start is greater than end.
    """
```

```
>>> print(numPrimes.__doc__)
```

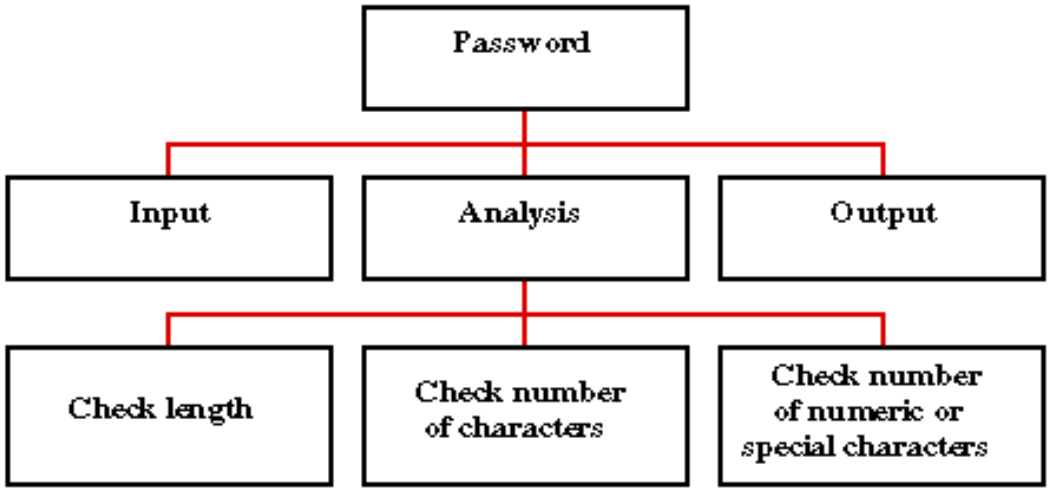
Top down design

Top-down design is an approach for deriving a modular design in which the overall design of a system is developed first, deferring the specification of more detailed aspects of the design until later steps.



Top down design

The goal of top-down design is that each module provides clearly defined functionality, which collectively provide all of the required functionality of the program.



Top down design

```
def getYear():

    """ Returns an integer value between 1800-2099, inclusive, or  -1."""
```

```
def leapYear(year):

    """ Returns True if provided year a leap year, otherwise returns False."""
```

```
def dayOfWeekJan1(year, leap_year):

    """ Returns the day of the week for January 1 of the provided year.

        year must be between 1800 and 2099. leap_year must be True if
        year a leap year, and False otherwise.
    """
```

Python modules

A Python **module** is a file containing Python definitions and statements. The Python Standard Library contains a set of predefined standard (built-in) modules.

LET'S TRY IT

Create a Python module by entering the following in a file name `simple.py`. Then execute the instructions in the Python shell as shown and observe the results.

```
# module simple
print('module simple loaded')

def func1():
    print('func1 called')

def func2():
    print('func2 called')
```

```
>>> import simple
???
>>> simple.func1()
???

>>> simple.func2()
???
```

Modules and namespaces

A namespace provides a context for a set of identifiers. Every module in Python has its own namespace. A name clash is when two otherwise distinct entities with the same identifier become part of the same scope.

```
# module1
def double(lst):

    """Returns a new list with each
       number doubled, for example,
       [1, 2, 3] returned as [2, 4, 6]
    """
```

```
# module2
def double(lst):

    """Returns a new list with each
       number duplicated, for example,
       [1, 2, 3] returned as
       [(1, 1), (2, 2), (3, 3)]
    """
```

```
import module1
import module2

# main
  .
  num_list = [3, 8, 14]
  result = double(num_list)
  .
```

ambiguous reference for identifier `double`

Modules and namespaces

```
import module1
import module2

# main
   ans1 = module1.double(...)
   ans2 = module2.double(...)
```

references function double from module1's namespace

references function double from module2's namespace

LET'S TRY IT

Enter each of the following functions in their own modules named `mod1.py` and `mod2.py`. Enter and execute the following and observe the results.

```
# mod1
def average(lst):
    print('average of mod1 called')

# mod2
def average(lst):
    print('average of mod2 called')
```

```
>>> import mod1, mod2
>>> mod1.average([10, 20, 30])
???
>>> mod2.average([10, 20, 30])
???
>>> average([10, 20, 30])
???
```

From of import

With the from-import form of import, imported identifiers become part of the importing module's namespace. Because of the possibility of name clashes, import modulename is the preferred form of import in Python.

```
from modulename import something
```

```
(a) from modulename import func1, func2
(b) from modulename import func1 as new_func1
(c) from modulename import *
```

Module private variables

In Python, all the variables in a module are “public,” with the convention that variables beginning with an two underscores are intended to be private.

```
class privateExample(object):
    def __init__(self):
        self.foo = 'Hello World'
    def __capital(self):
        print self.foo.upper()
    def lower(self):
        print self.foo.lower()
```

Module loading and execution

When a module is loaded, a compiled version of the module with file extension .pyc is automatically produced. When using the Python shell, an updated module can be forced to be reloaded and recompiled by use of the reload() function.

LET'S TRY IT

Create the following Python module named `simplemodule`, import it, and call function `displayGreeting` as shown from the Python shell and observe the results.

```
# simplemodule

def displayGreeting():
    print('Hello World!')
```

```
>>> import simplemodule
>>> simplemodule.displayGreeting()
```

Modify module `simplemodule` to display `'Hey there world!'`, import and again execute function `displayGreeting` as shown. Observe the results.

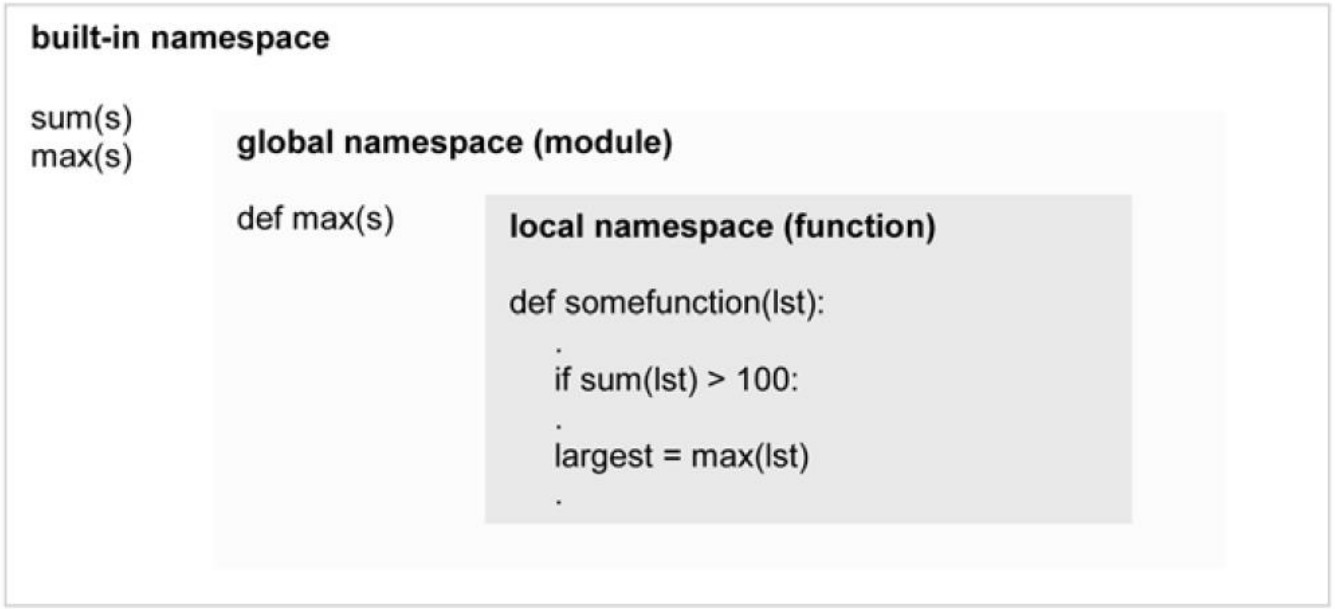
```
>>> import simplemodule
>>> simplemodule.displayGreeting()
```

Finally, reload the module as shown and again call function `displayGreeting`.

```
>>> reload(simplemodule)
>>> simplemodule.displayGreeting()
???
```

Local, global and Build Namespaces

At any given point in a Python program's execution, there are three possible namespaces referenced ("active")—the built-in namespace, the global namespace, and the local namespace.



Local, global and Build namespaces