Computational Thinking with Programming

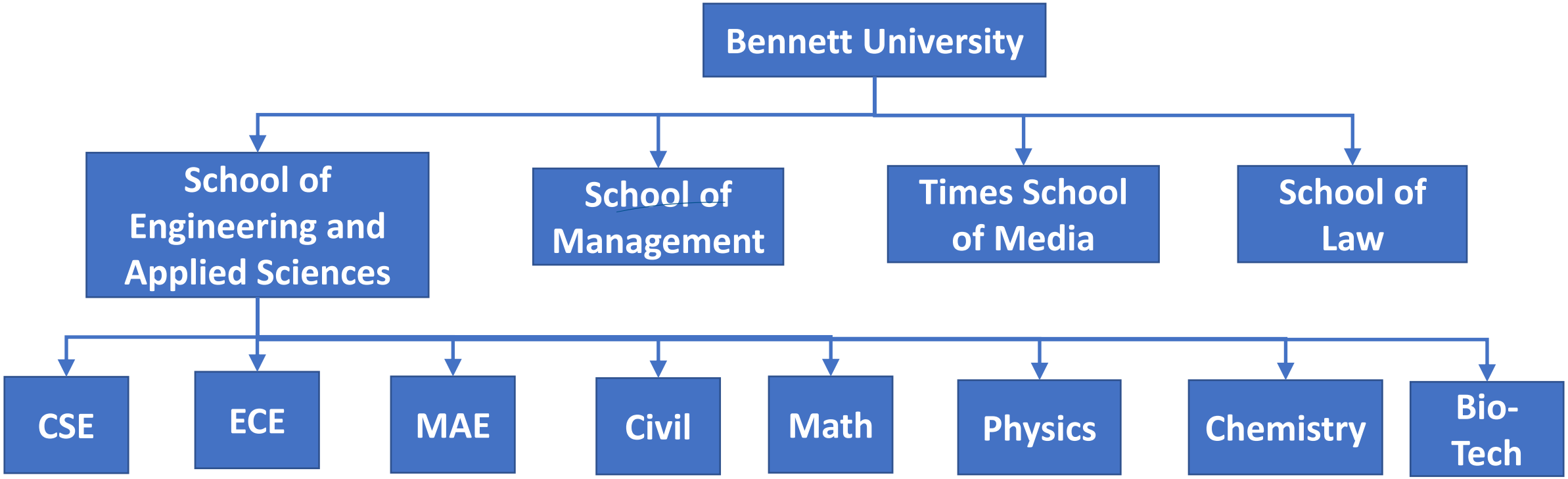
@cse_bennett @csebennett

Lecture Contents

- Modules
- Top-Down Approach
- Python Modules

Modules

- We can make complex systems more manageable by designing them as a set of subsystems, or modules.
- **Example:** Bennett University System



What is Module ?

- Programs are designed as a collection of modules.
- ***The term "module," broadly speaking, refers to the design and/or implementation of specific functionality to be incorporated into a program***.
- While an individual function may be considered a module, modules generally consists of a collection of functions (or other entities).

Advantages of Modules:

- 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

- Every module needs to provide a specification of how it is to be used, which is referred to as the module's **interface**.
- Any program code making use of a given module is called a **client** of the module.
- A module's specification should be sufficiently *clear* and *complete* so that its clients can effectively utilize it.
- A **docstring** is a string literal denoted by triple quotes used in Python for providing the specification of certain program elements.
- **Example:**

```
def numprimes(start, end):

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

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

MCQs

1. Which of the following is not an advantage in the use of modules in software development?
   a) Provides a natural means of dividing up programming tasks.
   b) Provides a means of reducing the size of a program.
   c) Provides a means for the reuse of program code.
   d) Provides a means of separately testing individual parts of a program.
   e) Provides a means of integrating parts of a program during testing.
   f) Facilitates the modification of specific program functionalities.
2. A specification of how a particular module is used is called the module's_____.
3. Program code that makes use of a given module is called a _____ of the module.
4. Indicate which of the following are true. A docstring in Python is
   a) A string literal denoted by triple or double quotes.
   b) A means of providing specification for certain program elements in Python.
   c) A string literal that may span more than one line.

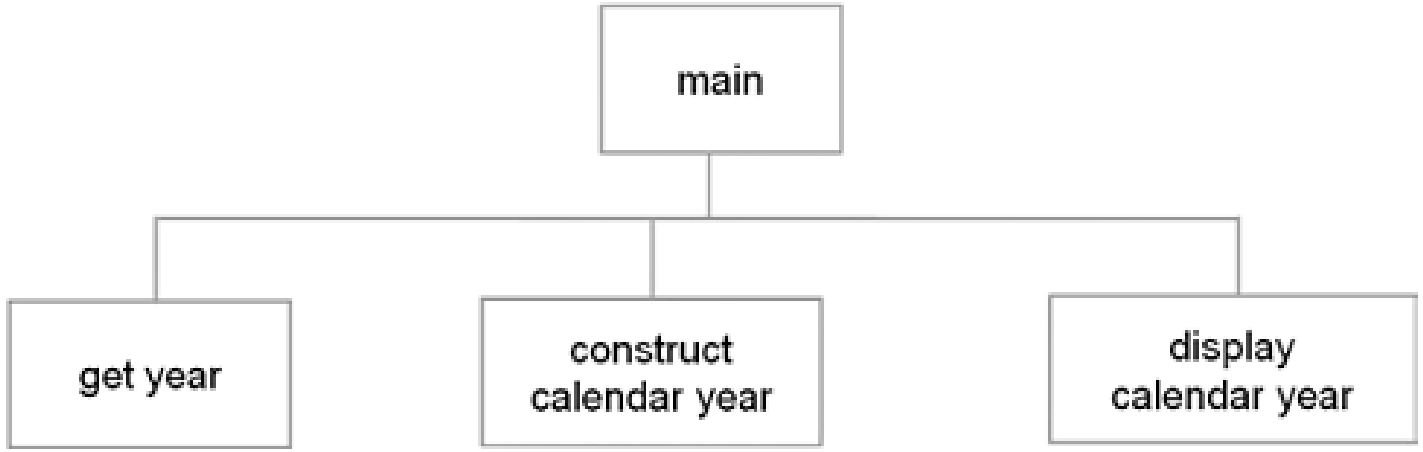
MCQs: Answer

1. Which of the following is not an advantage in the use of modules in software development?
   a) Provides a natural means of dividing up programming tasks.
   b) Provides a means of reducing the size of a program.
   c) Provides a means for the reuse of program code.
   d) Provides a means of separately testing individual parts of a program.
   e) Provides a means of integrating parts of a program during testing.
   f) Facilitates the modification of specific program functionalities.
2. A specification of how a particular module is used is called the module's **Interface**.
3. Program code that makes use of a given module is called a **client** of the module.
4. Indicate which of the following are true. A docstring in Python is
   a) A string literal denoted by triple or double quotes.
   b) A means of providing specification for certain program elements in Python.
   c) A string literal that may span more than one line.

Top-Down Design

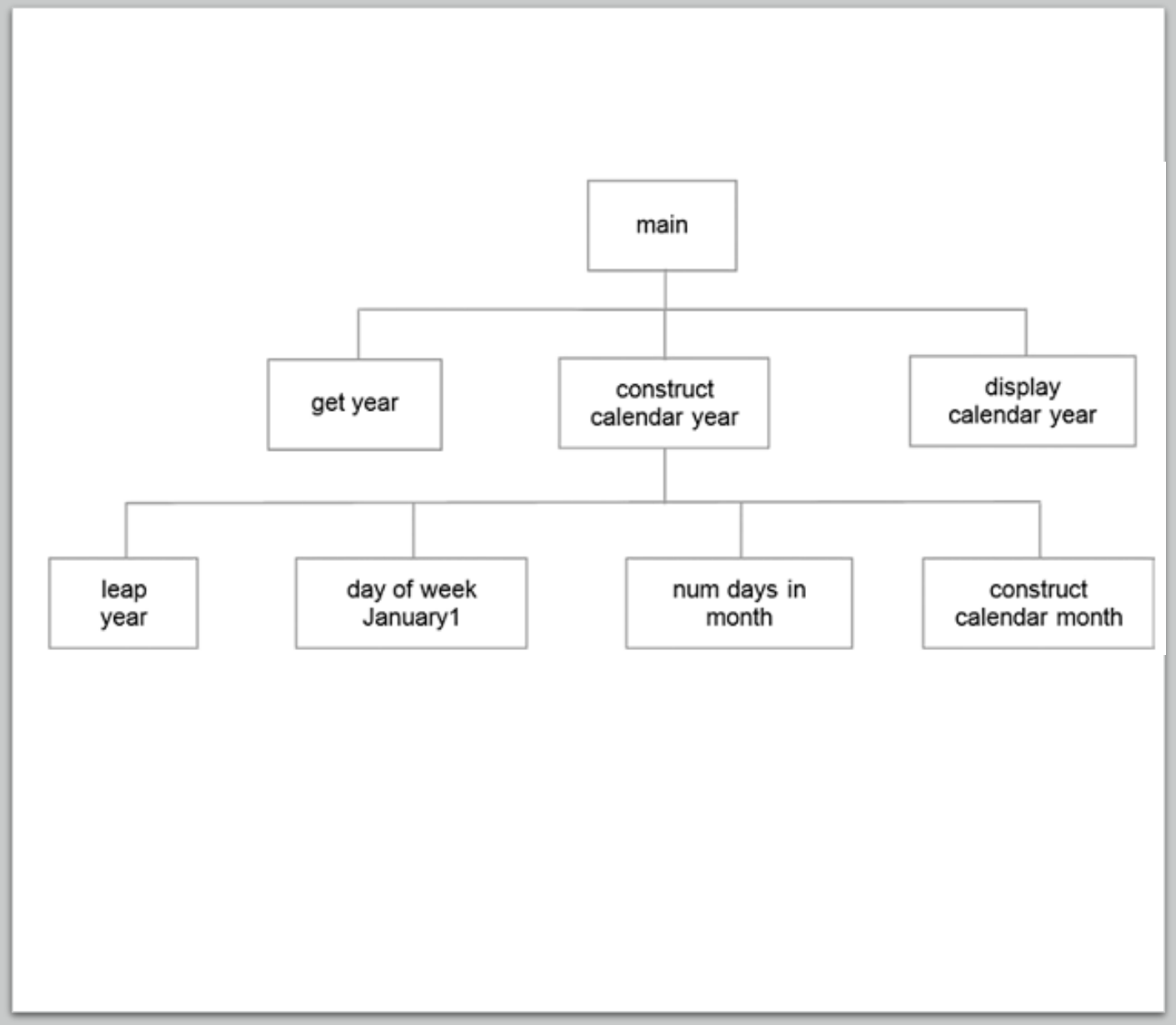
- **Top-down design** is an approach for deriving a modular design in which the overall design of a system is developed first, deferring the specification of more detailed aspects of the design until later steps.

- **Example: Developing a Modular Design of the Calendar Year Program**



Second Stage of Modular Design

- The goal of modular design is that each module provides clearly defined functionality, which collectively provide all of the required functionality of the program.
- Modules *get year* and *display calendar year* are not complex enough to require further breakdown.
- On the other hand, Module *construct calendar year has to do* most of the work, therefore, further broken down.



Specification of the Calendar Year Program Modules

- The modular design of the calendar year program provides a high-level view of the program
- Since each module is to be implemented as a function, we need to specify the details of each function.
- **Example:**
  - For each function it needs to be decided if it is a value-returning function or a non-value-returning function;
  - what parameters it will take;
  - what results it will produce.

- This stage of the design provides sufficient detail from which to implement the program. The main module provides the overall construction of the program.

MCQs

1. In top-down design (select one),
   a) The details of a program design are addressed before the overall design.
   b) The overall design of a program is addressed before the details.
2. All modular designs are a result of a top-down design process.
   a) TRUE
   b) FALSE
3. In top-down design, every module is broken down into the same number of submodules.
   a) TRUE
   b) FALSE
4. Which of the following advantages of modular design apply to the design of the calendar program.
   a) Provides a means for the development of well-designed programs.
   b) Provides a natural means of dividing up programming tasks.
   c) Provides a means of separately testing individual parts of a program.

MCQs: Answer

1. In top-down design (select one),
   a) The details of a program design are addressed before the overall design.
   b) The overall design of a program is addressed before the details.
2. All modular designs are a result of a top-down design process.
   a) TRUE
   b) FALSE
3. In top-down design, every module is broken down into the same number of submodules.
   a) TRUE
   b) FALSE
4. Which of the following advantages of modular design apply to the design of the calendar program.
   a) Provides a means for the development of well-designed programs.
   b) Provides a natural means of dividing up programming tasks.
   c) Provides a means of separately testing individual parts of a program.

Python Module

- A **Python module** is a file containing Python definitions and statements.
- A module allows you to logically organize your Python code.
- A module can define functions, classes, variables and can also include runnable code.
- **Module Creation and Use:** To create a module just save the code you want in a file with the file extension .py.
- We can use the module we just created, by using the import statement.

Example: Module Creation

```
#mymodule.py
def greeting(name):
  print("Hello, " + name)
```

Example: Using a Module

```
import mymodule
mymodule.greeting(“Bennett")
```

Module Creation and Use: Variables in Module

- The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc.).

Module Creation: Save this code in the file mymodule.py

```
person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

Module Use: Import the module named mymodule, and access the person1 dictionary

```
import mymodule
a = mymodule.person1["age"]
print(a)
```

Module Use: You can create the alias or rename the module while using.
Create an alias for mymodule called mx

```
import mymodule as mx
a = mx.person1["age"]
print(a)
```

Module Creation and Use: Example 2

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')
```

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

Note: When using a function from a module, use the syntax: *module_name.function_name*.

Library Modules

- The Python Standard Library contains a set of predefined standard (built-in) modules, which you can import whenever you want.

Example: Import and use the platform module:

```
import platform
x = platform.system()
print(x)
```

- There is a built-in function to list all the function names (or variable names) in a module, *i.e.* dir() function.

Example: List all the defined names belonging to the platform module:

```
import platform
x = dir(platform)
print(x)
```

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.
- Namespaces provide a means for resolving such problems.

```
# 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: Use of Fully Qualified Function Names

- Two instances of identifier *double*, each defined in their own module, are distinguished by being fully qualified with the name of the module in which each is defined: **module1.double** and **module2.double.**
- **Example:**

```
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])
???
```

Example:

Importing Modules

- In Python, the main module of any program is identified as the first ("top-level") module executed.
- With the import ***modulename*** form of import in Python, the namespace of the imported module becomes available to, but does not become part of, the namespace of the importing module.
- Therefore, the identifiers of the imported module, must be fully qualified (prefixed with the module's name) when accessed.

LET'S TRY IT

Enter the following into the Python shell and observe the results.

```
>>> factorial(5)
???

>>> math.factorial(5)
???
```

```
>>> import math
>>> factorial(5)
???

>>> math.factorial(5)
???
```

Import From Module

- You can choose to import only parts from a module, by using the from keyword.
- **Syntax:** ***from*** *modulename* ***import*** *something*
- *Something* can be a list of identifiers, a single renamed identifier, or an asterisk:

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

- (a), only identifiers func1 and func2 are imported.
- (b), only identifier func1 is imported, renamed as new_func1 in the importing module.
- (c), all of the identifiers are imported, except those which are private in module.

Import From Module: Example

- The module named mymodule has one function and one dictionary:

```
# mymodule
def greeting(name):
  print("Hello, " + name)

person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

- **Import only the person1 dictionary from the module:**

```
from mymodule import person1

print (person1["age"])
```

Note: *When using import* ***modulename****, the namespace of the imported module does not become part of the namespace of the importing module.*

Note: *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.*

Import From Module: Example

```
# module somemodule

def func1(n):
  return n * 10

def func2(n1, n2):
  return n1 * n2
```

```
# ---- main

from somemodule import *

# NAMESPACES AND FUNCTION USE

def func2(n):
    return n * n                # definition of func2 masks imported func2

print(func1(8))                 # outputs 80 (func1 of somemodule called)
print(somemodule.func1(8))      # NameError: name 'somemodule' is not defined

print(func2(5))                 # outputs 25 (func2 of MAIN called)
print(func2(3, 8))              # TypeError: func2() takes exactly 1 argument

print(somemodule.func2(3, 8)) # NameError: name 'somemodule' is not
                                  defined
```

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.
- Public is accessible by any other module that imports it, whereas private is not accessible in form of ***from modulename import ****.*
- Python does not provide any means for preventing access to variables or other entities meant to be private.
- There is a convention that names beginning with two underscores (__) are intended to be private.

Example: **# mymodule**

```
def __greeting(name):
    print("Hello, " + name)
```

Module Loading and Execution

- Each imported module of a Python program needs to be located and loaded into memory.
- 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, i.e. *installation-specific* (e.g., C:\Python32\Lib)
- The module search path is stored in the system module sys as the **sys.path** variable. The sys.path variable contains the current directory, ***PYTHONPATH***, and the installation-dependent default.
- If the program still does not find the module, an error (***ImportError*** exception) is reported.

Lets 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`.

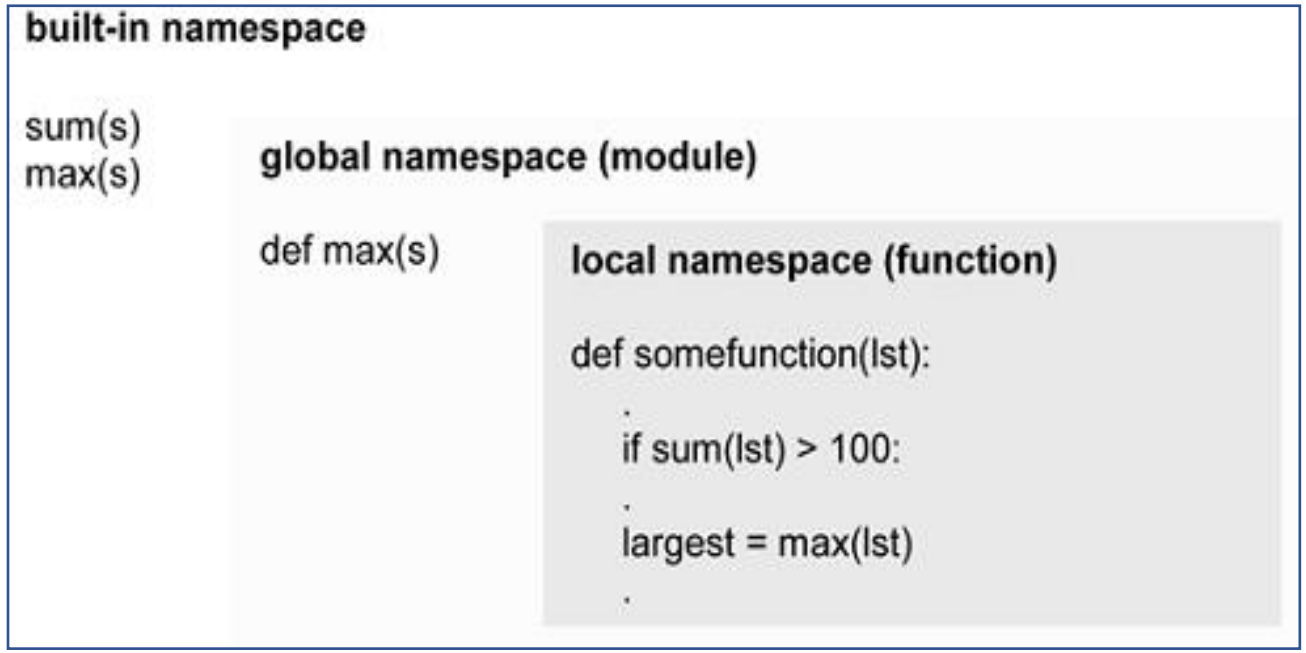
```
>>> reload(simplemodule)
>>> simplemodule.displayGreeting()
???
```

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.

Local, Global, and Built-in Namespaces in Python

- 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**.
- The **built-in** namespace contains the names of all the built-in functions, constants, and so on, in Python.
- The **global** namespace contains the identifiers of the currently executing module.
- And the **local** namespace is the namespace of the currently executing function (if any).

Example:



max([4, 2, 7, 1, 9, 6]) → 9 (built-in function max)
max([4, 2, 7, 1, 9, 6]) → 4 (programmer-defined function max)

Built-In, Global: Example

```
# grade_calc module

def max(grades):
    largest = 0

    for k in grades:
        if k > 100:
            largest = 100
        elif k > largest:
            largest = k

    return largest

def grades_highlow(grades):
    return (min(grades), max(grades))
```

```
# classgrades (main module)

from grade_calc import *

class_grades = [86, 72, 94, 102, 89, 76, 96]

low_grade, high_grade = grades_highlow(class_grades)
print('Highest adjusted grade on the exam was', high_grade)
print('Lowest grade on the exam was', low_grade)

print('Actual highest grade on exam was', max(class_grades))
```

```
Highest adjusted grade on the exam was 100
Lowest grade on the exam was 72
The actual highest grade on the exam was 100
>>>
```

Local, Global Example

- A Python statement can access variables in a *local namespace* and in the *global namespace*.
- If a local and a global variable have the same name, the local variable shadows the global variable.

```
Example1:
Money = 2000
def AddMoney():
       Money = Money + 1
print Money
AddMoney()
print Money
```

Output: ?

Local, Global: Example

- In order to assign a value to a global variable within a function, you must first use the global statement.

Example1:

```
Money = 2000
def AddMoney():
        Money = Money + 1
print Money
AddMoney()
print Money
```

Output: UnboundLocalError

Example2:

```
Money = 2000
def AddMoney():
        global Money
        Money = Money + 1
print Money
AddMoney()
print Money
```

Output: 2000
2001

LET'S TRY IT

Enter the following in the Python shell:

```
>>> sum([1, 2, 3])
???

>>> def sum(n1, n2, n3):
        total = n1 + n2 + n3

        return total

>>> sum([1, 2, 3])
???

>>> sum(1, 2, 3)
???
```

Create a file with the following module:

```
# module max_test_module
def test_max():
    print 'max =', max([1, 2, 3])
```

Create and execute the following program:

```
import max_test_module

def max():
    print('max:local namespace called')

print(max_test_module.test_max())
```

Exercise:

MCQs

1. Any initialization code in a Python module is only executed once, the first time that the module is loaded.
   a) True
   b) False
2. With the "import *moduleName*" form of import, any utilized entities from the imported module must be prefixed with the module name.
   a) True
   b) False
3. When importing modules, all Python Standard Library modules must be imported before any programmer-defined modules, otherwise a runtime error will occur.
   a) True
   b) False
4. By convention, variables names in a module beginning with two _____ characters
5. If a particular module is imported more than once in a program, the Python interpreter will ensure that the module is only loaded and executed the first time that it is imported.
   a) True
   b) False
6. The _____command can be used to force the reloading of a given module, useful for when working interactively in the Python shell.
7. The three active namespaces that may exist during the execution of any given Python program are the _____, _____ and _____ namespaces.

MCQs: Answers

1. Any initialization code in a Python module is only executed once, the first time that the module is loaded.
   a) True
   b) False
2. With the "import *moduleName*" form of import, any utilized entities from the imported module must be prefixed with the module name.
   a) True
   b) False
3. When importing modules, all Python Standard Library modules must be imported before any programmer-defined modules, otherwise a runtime error will occur.
   a) True
   b) False
4. By convention, variables names in a module beginning with two **underscores** characters
5. If a particular module is imported more than once in a program, the Python interpreter will ensure that the module is only loaded and executed the first time that it is imported.
   a) True
   b) False
6. The **reload** command can be used to force the reloading of a given module, useful for when working interactively in the Python shell.
7. The three active namespaces that may exist during the execution of any given Python program are the **built-in**, **global** and **local** namespaces.

Thank You