\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**CSC121 PYTHON Programming**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Lesson 12 Modular Design

# **Objectives**

In this lesson, students will learn:

- How to use top-down approach to design a program

- How to use docstring for function specification

- How to create modules in Python

- How to import Python modules

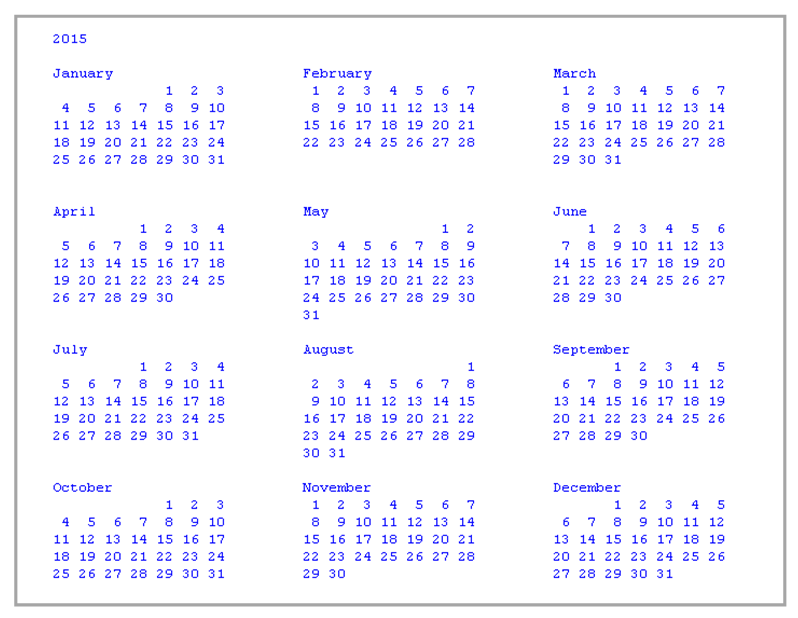
# **12.1 Overview**

We have learned a few of things about programming. We know how to create and use variables, how to use control structures to control the flow of the program, how to use lists to handle large amount of data, and how to write and use functions. All the programs we have written are short. It is time to use all these techniques to write a longer program.

# **12.2 The Calendar Year Program**

The textbook uses a calendar year program to demonstrate how to design and write a longer program. You can find this program in chapter 7 of the textbook.

The goal of the program is to display a calendar year for any year between 1800 and 2099, inclusive. Figure 7-19 of the textbook shows how the calendar should look like if the user chooses to see the calendar of 2015:



Let’s think about how to write this program. First of all, we need to know which year we need to display. This problem is very easy to solve. We simply ask the user to enter the year he wants to see.

Next we need to know whether that year is a leap year or not. This will determine whether there are 28 or 29 days in February.

Third, we need to know what day of the week January 1 of that year falls on. Once we figure this out, we can determine the day of the week of each day in the rest of the whole year.

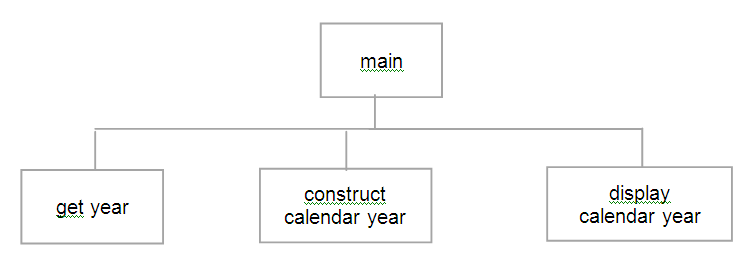
Once we have the three things above determined, the rest is just figuring out how to display the months.

# **12.3 Top-Down Design**

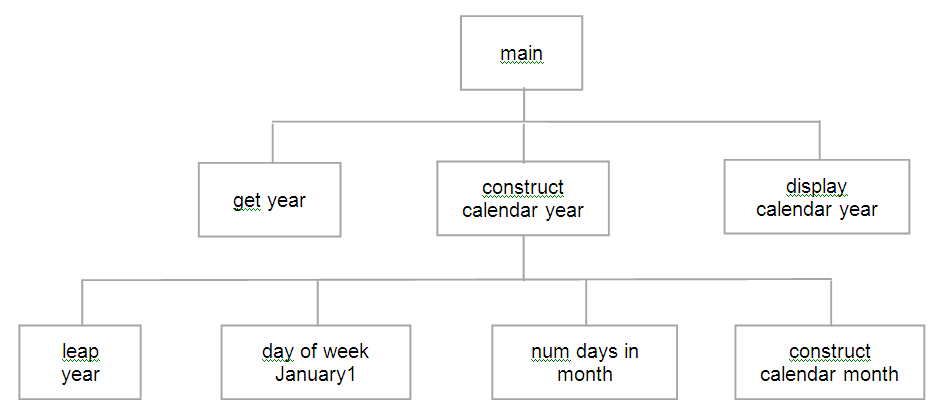
As mentioned before, an important aspect of well-designed software is that programs are designed as a collection of modules. The term “module”, broadly speaking, refers to the design and/or implementation of specific functionality to be incorporated into a program. Modular design allows large programs to be broken down into manageable size parts, in which each part provides a clearly specified capacity. It aids the software development process by providing an effective way of separating programming tasks among various individuals or teams. It allows modules to be individually developed and tested, and eventually integrated as a part of a complete system. Modular design also facilitates program modification since the code responsible for a given aspect of the software is contained within specific modules, and not distributed throughout the program.

One method of deriving a modular design is called **top-down design**. In this approach, the overall design of a system is developed first, deferring the specification of more detailed aspects of the design until later steps.

Let’s use a top-down design approach to develop a modular design for the calendar year program. The three overall steps of the program are getting the requested year from the user, creating the calendar year structure, and displaying the year. This is depicted in Figure 7-6 of the textbook:



We then consider whether any of these modules needs to be further broken down. Making such a decision is more of an art than a science. The goal of modular design is that each module provides clearly defined functionality, which collectively provides all of the required functionality of the program. Modules “get year” and “display calendar year” are not complex enough to require further breakdown. Module “construct calendar year”, on the other hand, is where most of the work is done, and is therefore further broken down. Figure 7-7 of the textbook contains the modules of this next design step:



In order to construct a calendar year, it must be determined whether the year is a leap year, what day of the week January 1 of that year falls on, and how many days are in each month (accounting for leap year). Thus, modules “leap year”, “day of week January 1” and “numbers of days in month” are added as submodules of module “construct calendar year”. The calendar month for each of the twelve months must be then individually constructed, handled by module “construct calendar month”.

# **12.4 Designing the Functions**

The modular design of the calendar year program provides a high-level view of the program. However, there are many issues yet to resolve in the design. Since each module is to be implemented as a function, we need to specify the details of each function, including what parameters it will take, what results it will produce, and whether it will return any values. We are going to write eight functions in this program.

|  |  |
| --- | --- |
| Function | Specification |
| getYear() | This function gets the year the user wants to see. It has no parameters, and it returns an integer value between 1800-2099, inclusive, or -1 to terminate the program |
| leapYear(year) | This function determines whether the year is a leap year or not. It has one parameter: year must be between 1800 and 2099. It returns True if provided year is a leap year, otherwise returns False. |
| dayOfWeekJan1(year, leap\_year) | This function returns the day of the week for January 1 of the provided year. It has two parameters: year must be between 1800 and 2099, leap\_year must be True if year a leap year, and False otherwise. |
| numDaysinMonth (month\_num, leap\_year) | This function returns the number of days in a given month. It has two parameters: month\_num must be in the range 1 – 12, inclusive; leap\_year must be True if year a leap year, and False otherwise. |
| constructCalMonth (month\_num, first\_day\_of\_month, num\_days\_in\_month) | This function returns a formatted calendar month for display on the screen. It has three parameters: month\_num must be in the range 1 – 12, inclusive; first\_day\_of\_month must be in the range 0 – 6 (1-Sun, 2-Mon, …, 0-Sat); num\_days\_in\_month must be in the range 1 – 31. Return a list of strings of the form [month\_name, week1, week2, week3, week4, week5, week6] |
| constructCalYear(year) | This function returns a formatted calendar year for display on the screen. It has one parameter: year must be between 1800 and 2099, inclusive. Return a list of lists of strings of the form [year, month1, month2, month3, …, month12] |
| dsplayCalendar (calendar\_year) | The function displays the provided calendar\_year on the screen three months across. It has one parameter: calendar\_year should be in the form of a list of lists of strings of the form [year, month1, month2, month3, …, month12] |
| main() | This function calls the getYear function to obtain the year the user wants to see, the constructCalYear function to construct the calendar year, and the displayCalendar function to display the calendar year |

# **12.5 Coding the Functions**

We are going to write the code for every function in this project. Let’s start with the getYear function. The following is the Python code:

**def** getYear():  
year = int(input(**'Enter year (yyyy) (-1 to quit): '**))  
 **while** (year < 1800 **or** year > 2099) **and** year != -1:  
 print (**'INVALID INPUT - Year must be between 1800 and 2099'**)  
 year = int(input(**'Enter year (yyyy) (-1 to quit): '**))  
   
 **return** year

This function asks the user to enter the year he wants to see or -1 to quit. The year must be between 1800 and 2099, inclusive. An input validation loop is used to check whether the year is out of range.

Before we move on to the next function, let’s add some documentation to this function. When a programming project gets larger, sometimes multiple programmers are involved in coding. Better documentation becomes essential to avoid misunderstandings between programmers in the specification of a function. At the minimal, there should be at least a one-line specification of what the function is designed to do. In Python, the convention is using a triple double-quoted string. The following is an example:

*""" Returns a year between 1800-2099, inclusive, or the value -1 """*

In Python, this is called **docstring**. It is typically inserted between the function header and the function body. The following is the definition of getYear with docstring:

**def** getYear():

*""" Returns a year between 1800-2099, inclusive, or the value -1 """* year = int(input(**'Enter year (yyyy) (-1 to quit): '**))  
 **while** (year < 1800 **or** year > 2099) **and** year != -1:  
 print (**'INVALID INPUT - Year must be between 1800 and 2099'**)  
 year = int(input(**'Enter year (1-12): '**))  
   
 **return** year

Blank lines are inserted before and after the docstring and also before the return statement to make the code easier to read.

Next, let’s write the code for the leapYear function:

**def** leapYear(year):

*""" Returns True if year a leap year, otherwise returns False. """* **if** (year % 4 == 0) **and** (**not** (year % 100 == 0) **or** (year % 400 == 0)):  
 leap\_year = **True  
 else**:  
 leap\_year = **False  
  
 return** leap\_year

A year is a leap year if it fulfills both of the following two requirements:

1. It is divisible by 4, and
2. It is either indivisible by 100 or it is divisible by 400

An if else statement is written to test these requirements and return True if the year is a leap year or False otherwise.

Next, let’s write the code for the dayOfWeekJan1 function.

**def** dayOfWeekJan1(year, leap\_year):

*""" Returns the day of the week for January 1 of a given year.  
  
 year must be between 1800 and 2099. leap\_year must be True if  
 year a leap year, and False otherwise. """* century\_digits = year // 100  
 year\_digits = year % 100  
 value = year\_digits + (year\_digits // 4)  
  
 **if** century\_digits == 18:  
 value = value + 2  
 **elif** century\_digits == 20:  
 value = value + 6  
  
 *# adjust for leap years* **if not** leap\_year:  
 value = value + 1  
   
 *# return first day of month for Jan 1* **return** (value + 1) % 7

This function determines the day of the week of January 1 of the calendar year. The algorithm is described in page 105, 106 and 149 of the textbook. There are 52 weeks and 1 day in a regular year. Therefore, the day of the week of January 1 shifts one day if the previous year is not a leap year and 2 days if the previous year is a leap year. This function uses the day of January 1 in the first year of that century as the starting point and uses the year digits to calculate how many days it should shift down. January 1 of 1800 is Wednesday. January 1 of 1900 is Monday. January 1 of 2000 is Friday. For example, suppose we want to display the year 1903. January 1 of 1900 is Monday. The year digits are 03. The day of the week of January 1 shifts by 3 days to Thursday. Let’s look at another example. Suppose we want to display the year 1905. January 1 of 1900 is Monday. The year digits are 05. Since there is one leap year every four years, the day of the week of January 1 shifts by 6 days (i.e. 05 + 1 = 6) to Sunday. An integer value from 0 to 6, which represents the day of the week, is returned.

Sometimes we need a function specification that is longer than one single line. In that case, the docstring can expand to multiple lines. The convention is putting an overall description of what the function does in the first line. Leave the second line blank, and then add further specification starting from the third line.

Next, let’s write the code for the numDaysInMonth function:

**def** numDaysInMonth(month\_num, leap\_year):  
 *""" Returns the number of days in a given month.  
  
 month\_num in the range 1-12, inclusive.  
 leap\_year True if month in a leap year, otherwise False. """* num\_days\_in\_month = (31,28,31,30,31,30,31,31,30,31,30,31)  
  
 *# special check for February in leap year* **if** (month\_num == 2) **and** leap\_year:  
 num\_days = 29  
 **else**:  
 num\_days = num\_days\_in\_month[month\_num-1]  
 **return** num\_days

This function returns the number of days in a given year. The number of days for each month in the year is stored in a tuple. If the given year is a leap year, 29 instead of 28 is retuned for February.

Next, let’s write the code for the constructCalMonth function:

**def** constructCalMonth(month\_num, first\_day, num\_days\_in\_month):  
 *""" Returns a formatted calendar month for display on the screen.  
  
 month\_num in the range 1-12, inclusive.  
 first\_day in the range 0-6 (1-Sun, 2-Mon, ..., 0-Sat)  
   
 Returns a list of string values of the form,  
 [month\_name, week1, week2, week3, week4, ...] """  
  
 # init* empty\_str = **''** blank\_col = format(**' '**, **'3'**)  
 blank\_week = format(**' '**, **'21'**)  
 month\_names = (**'January'**, **'February'**, **'March'**, **'April'**, **'May'**, **'June'**,  
 **'July'**, **'August'**, **'September'**, **'October'**, **'November'**,  
 **'December'**)  
  
 calendar\_month = [**' '** + format(month\_names[month\_num - 1],**'<20'**)]  
 current\_day = 1  
 current\_col = 1  
 calendar\_week = **''** *# init starting column* **if** first\_day == 0:  
 starting\_col = 7  
 **else**:  
 starting\_col = first\_day  
  
 *# add any needed leading spaces for first week of month* **while** current\_col < starting\_col:  
 calendar\_week = calendar\_week + blank\_col  
 current\_col = current\_col + 1  
   
 *# construct month for proper number of days* **while** current\_day <= num\_days\_in\_month:  
  
 *# store day of month in field of length 3* calendar\_week = calendar\_week + format(str(current\_day),**'>3'**)  
  
 *# append new week to month if at end of week* **if** current\_col == 7:  
 calendar\_month = calendar\_month + [calendar\_week]  
 calendar\_week = empty\_str  
 current\_col = 1  
 **else**:  
 current\_col = current\_col + 1  
   
 current\_day = current\_day + 1  
  
 *# if there is a final week, append to constructed month* **if** calendar\_week != empty\_str:  
 calendar\_month = calendar\_month + [calendar\_week]  
   
 **return** calendar\_month

This is the longest function in the whole program. It creates a list of strings. Each string in this list is the dates of a week in that month. Each day occupies 3 spaces. The whole week occupies 21 spaces. Blank spaces are inserted at the beginning of the first week if the first day of the month is not Sunday.

Next, let’s write the code for the constructCalYear function:

**def** constructCalYear(year):

*""" Returns a formatted calendar year for display on the screen.  
  
 year in the range 1800-2099, inclusive  
 Returns a list beginning with the year, followed by  
 twelve constructed months  
   
 [year, month1, month2, month3, ..., month12] """  
  
 # init* leap\_year = leapYear(year)  
 first\_day\_of\_month = dayOfWeekJan1(year, leap\_year)  
 calendar\_year = [year]  
  
 *# construct calendar from twelve constructed months* **for** month\_num **in** range(1,13):  
 num\_days\_in\_month = numDaysInMonth(month\_num, leap\_year)  
   
 calendar\_year.append(constructCalMonth(  
 month\_num,  
 first\_day\_of\_month,  
 num\_days\_in\_month))  
   
 first\_day\_of\_month = (first\_day\_of\_month + num\_days\_in\_month) % 7  
  
 **return** calendar\_year

This function returns a formatted calendar year for display on the screen. It calls the constructCalMonth function twelve times, each for a month, and appends the return month list to form a list for the whole year.

Next, let’s write the code for the displayCalendar function:

**def** displayCalendar(calendar\_year):

*""" Displays a calendar\_year on the screen three months across. """  
  
 # init* month\_separator = format(**' '**, **'8'**)  
 blank\_week = format(**' '**, **'21'**)  
  
 *# display year* print(**'\n'**, calendar\_year[0])  
  
 *# display months three across* **for** month\_index **in** [1,4,7,10]:  
  
 *# init* week = 1  
 lines\_to\_print = **True  
  
 while** lines\_to\_print:  
   
 *# init* lines\_to\_print = **False** *# print weeks of months side-by-side* **for** k **in** range(month\_index, month\_index + 3):  
 **if** week <= len(calendar\_year[k]):  
 week\_dates = calendar\_year[k][week-1]  
 print(week\_dates + blank\_week[len(week\_dates):], end=**''**)  
 lines\_to\_print = **True  
 else**:  
 print(blank\_week, end=**''**)  
 print(month\_separator, end=**''**)  
   
 *# move to next screen line* print()  
  
 *# increment week* week = week + 1

This function uses nested loops to display the calendar. The outer loop iterates four times. Three months are displayed side-by-side in each iteration. An inner loop is used to display every week of the three months.

Finally, let’s write the code for the main function:

**def** main():  
  
 *""" Displays calendar of the years chosen by the user """*

*# initialization* terminate = **False** *# program greeting* print (**'This program will display a calendar for a given year'**)  
  
 *# continue to display calendar years until -1 entered* **while not** terminate:  
 year = getYear()  
  
 **if** year == -1:  
 terminate = **True  
 else**:  
 *# construct calendar* calendar\_year = constructCalYear(year)  
  
 *# display calendar* displayCalendar(calendar\_year)

This function calls the getYear function to get the user’s choice of year. It calls the constructCalYear function to construct the calendar and the displayCalendar function to display it. A loop is used to repeat this as many times as the user wants.

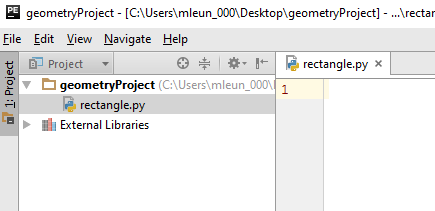
In summary, this calendar year example demonstrates how to design and code a non-trivial program. A top-down approach is used to create a hierarchy of tasks. Functions are written to implement these tasks.

# **12.6 Python Modules**

In programming, the term “module” refers to the design and/or implementation of specific functionality to be incorporated into a program. In other words, it is broadly used to mean “functional parts”. However, in the Python programming language, the term “module” has a very specific meaning. A **Python module** is a file containing function and class definitions and other statements. So far in every Python project, we always put every line of code in one single file. In larger Python projects, it is common to distribute the code to multiple Python files. Each Python file is a module. When we run the program, only one module is directly executed. This module is considered the *main module* of the program. If the main module needs to execute code in other modules, we need to add statements in the main module to import other modules. We will see exactly how this is done soon.

Suppose we are writing a program for geometric calculation. This program calculates area and perimeter of a rectangle or a circle. In addition of the main method, we are writing fourth addition functions: two functions that calculate the area and perimeter of a rectangle, and two functions that calculate the area and perimeter of a circle. We will create two modules: a rectangle module that contains functions about rectangles, and a circle module that contains functions about circles.

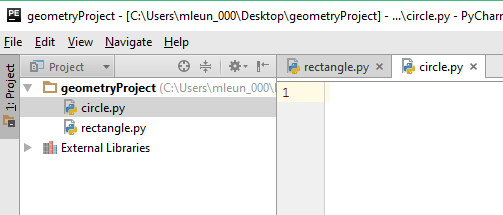
First of all, let’s create a Python project and name it geometryProject. Next, let’s write the rectangle module. To do that, add a Python file to the project and name the file rectangle.py. This file will contain the code of this module.



Next, let’s add the code of the area and perimeter functions to the rectangle module. In other words, type following code in the file rectangle.py:

**def** area(width, length):  
   
 *""" Returns area of rectangle with given length and width """* **return** width \* length  
  
  
**def** perimeter(width, length):  
   
 *""" Returns perimeter of rectangle with given length and width """* **return** 2 \* (width + length)

Next, let’s create a Python file circle.py in the project to contain functions about circles.

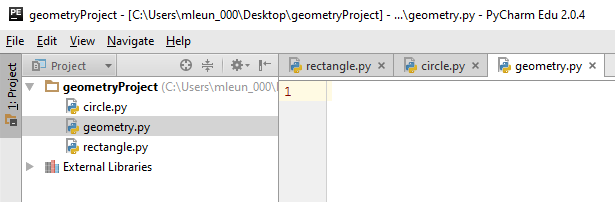


Next, let’s write the code for this circle module.

PI = 3.1416  
  
**def** area(radius):  
   
 *""" Returns area of circle with given radius """* **return** PI \* radius\*\*2  
  
  
**def** circumference(radius):  
   
 *""" Returns circumference of circle with given radius """* **return** 2 \* PI \* radius

There are two functions in this module. The function area calculates the area of a circle, while the function circumference calculates circumference. A variable PI is created globally before the definitions of these functions so its value can be accessed by these functions.

Next, let’s add a new Python file named geometry.py to the project.



This file is the main module of this project. Let’s add code to this file.

**import** circle *# import circle module***import** rectangle *# import rectangle module***def** main():  
  
 *""" Calculates area and perimeter of rectangle or circle """* print(**"This program calculates area and perimeter of a rectangle or circle."**)  
 choice = int(input(**"Enter 1 for rectangle or 2 for circle: "**))  
  
 **if** choice == 1:  
 length = float(input(**"Enter length of rectangle: "**))  
 width = float(input(**"Enter width of rectangle: "**))  
 rect\_area = rectangle.area(length, width)  
 rect\_perimeter = rectangle.perimeter(length, width)  
 print(**"Area:"**, rect\_area, **"Perimeter:"**, rect\_perimeter)  
  
 **elif** choice == 2:  
 radius = float(input(**"Enter radius of circle: "**))  
 circle\_area = circle.area(radius)  
 circle\_perimeter = circle.circumference(radius)  
 print(**"Area:"**, circle\_area, **"Perimeter:"**, circle\_perimeter)  
 **else**:  
 print(**"Invalid choice"**)  
  
main()

This file is the main module. It contains code we want the computer to execute directly when the program starts to run. In this example, we have only one function (i.e. the main function) defined in this module but it can have as many functions as necessary if what you want it to do in this main module is too complex to fit into one single function. At the top of this file, we have two statements to import the circle and rectangle modules into this main module. These two statements are necessary if you want to call functions defined in those two modules.

In the main function, the user is asked to choose either rectangle or circle. If rectangle is chosen, the user is asked to enter the length and width. The area and perimeter functions of the rectangle module are called with length and width passed to these functions as arguments. If circle is chosen, the user is asked to enter the radius. The area and circumference functions of the circle module are called with radius passed to these functions as an argument. Notice that when the function of an imported module is called, we need to use both the module name and the function name to specify which function of which module is called. For example, the following statement calls the area function of the rectangle module:

area = rectangle.area(length, width)

The following statement calls the area function of the circle module:

area = circle.area(radius)

If a function is called without module name, then the function is assumed to be defined in the local module (i.e. defined in the same file). For example, the following statement calls the main function defined in the local module:

main()

You may wonder why we bother to create and use multiple modules. Why don’t we simply put all the code in one single file? First of all, multiple modules are typically used only for larger programming projects. If there are only a handful of functions in a project, we probably will simply put all function definitions in one single module. The geometry example actually may be too short to use multiple modules. We just wanted to use a short and simple example to illustrate how to design and use multiple modules.

When we are working on a large project, we should plan to create multiple modules for at least two reasons. First, modules give better organization to the project by putting related functions into groups. This is particularly helpful when multiple programmers are working together on the same project because each programmer can be responsible for writing one or several modules. Second, modules allow us to reuse code in another project easier. For example, we can easily reuse the circle and rectangle modules in other programming projects about geometrical shapes.

It is a common practice in programming to reuse code already written. In fact, when you install Python in your computer, a set of commonly used modules, called **Python Standard Library**, are copied to the computer along with the Python interpreter. These modules contain functions and other tools that are frequently used in Python programs. When you use functions from these modules, you do not need to copy the code of these modules to your Python project. You only need to use import statements to import the modules you want. The Python interpreter will know where these module files are located and access the function definitions when those functions are called. Let’s look at one example.

The Python math module contains a number of functions for mathematics. This example uses the sqrt function to calculate the square root of a number.

**import** math  
  
x = float(input(**"Enter a number: "**))  
square\_root = math.sqrt(x)  
print(**"The square root of the number is"**, square\_root)

# **12.7 Namespaces**

In the previous section it is mentioned that when a function of an imported module is called, we need to specify both the module name and the function name. This is done to avoid potential name clash, which happens when two otherwise distinct entities with the same name become part of the same scope. In the geometry example, the names rectangle.area and rectangle.perimeter are used to call the area and perimeter functions of the rectangle module, while the names circle.area and circle.circumference are used to call the area and circumference functions of the circle module. In programming, these names are called *fully qualified names*, which are unambiguous names that specify which functions are referred to without regard of the context. In these fully qualified names, the first part, i.e. the module name, is the module’s **namespace**. A namespace is a container that provides a named context for a set of identifier. Namespaces enable programs to avoid potential name clashes by associating each identifier with the namespace from which it originates.

In Python, each module has its own namespace. This includes the names of all items in the module, including functions and *global variables*, i.e., variables defined within the module and outside the scope of any of its functions. For example, in the circle module, the variable PI is a global variable, which is defined outside any functions of the circle module and accessible to all functions in the module. In fact, we can use a fully qualified name to access the PI from the main module:

…

**elif** choice == 2:  
 print(**"Vale of PI in circulations:"**, circle.PI)   
 radius = float(input(**"Enter radius of circle: "**))  
 circle\_area = circle.area(radius)  
 circle\_perimeter = circle.circumference(radius)  
 print(**"Area:"**, circle\_area, **"Perimeter:"**, circle\_perimeter)

An extra statement is inserted in the main function of the main module. If the user chooses circle, the program displays the value of PI, which is a global variable in the circle module, by using the fully qualified name circle.PI. The following is a sample output:

This program calculates area and perimeter of a rectangle or circle.

Enter 1 for rectangle or 2 for circle: 2

Vale of PI in circulations: 3.1416

Enter radius of circle: 10

Area: 314.15999999999997 Perimeter: 62.832

In Python, there is a way for you not to use the fully qualified name to call functions in a different module. In the example we saw in the previous section, we use the following statements to import the circle and rectangle modules:

**import** circle *# import circle module***import** rectangle *# import rectangle module*

Let’s rewrite these statements like this:

**from** rectangle **import** perimeter  
**from** circle **import** circumference

Now there is no need to use fully qualified names when we call the perimeter function of the rectangle module and the circumference function of the circle module. The area function in these modules is a little tricky since there is a name clash. We will talk about how to call the area functions without using fully qualified names later.

These new import statements need some explanations. First, these statements only import the specified functions. Functions that are not specified are not imported. Therefore, the area functions in these modules are not imported and cannot be called in the main module.

The following is the modified code of the main module. Calculations of areas are removed temporarily.

**from** rectangle **import** perimeter  
**from** circle **import** circumference  
  
**def** main():  
  
 *""" Calculates perimeter of rectangle or circle """* print(**"This program calculates perimeter of a rectangle or circle."**)  
 choice = int(input(**"Enter 1 for rectangle or 2 for circle: "**))  
  
 **if** choice == 1:  
 length = float(input(**"Enter length of rectangle: "**))  
 width = float(input(**"Enter width of rectangle: "**))  
 rect\_perimeter = perimeter(length, width)  
 print(**"Perimeter:"**, rect\_perimeter)  
  
 **elif** choice == 2:  
 radius = float(input(**"Enter radius of circle: "**))  
 circle\_perimeter = circumference(radius)  
 print(**"Perimeter:"**, circle\_perimeter)  
  
 **else**:  
 print(**"Invalid choice"**)  
  
main()

Notice that when the perimeter function of the rectangle module is called, only the function name is used. The module name is not included in the call:

rect\_perimeter = perimeter(length, width)

Similarly, when the circumference function of the circle module is called, only the function name is used:

circle\_perimeter = circumference(radius)

The module name is not used in these calls because when the from-import statement is used to import functions from a module, the imported module’s namespace becomes part of the importing module’s namespace. Thus, imported identifiers are referenced without being fully qualified.

If you want to use the from-import statement to import all functions from a module, you do not have to list the names of all functions. You can use \* to include all functions:

**from** rectangle **import** \*  
**from** circle **import** \*

The statements above import all functions from the rectangle and circle modules.

The following is the modified code for the main module. Statements calculating areas are inserted back in the program.

**from** rectangle **import** \*  
**from** circle **import** \*  
  
**def** main():  
  
 *""" Calculates area and perimeter of rectangle or circle """* print(**"This program calculates area and perimeter of a rectangle or circle."**)  
 choice = int(input(**"Enter 1 for rectangle or 2 for circle: "**))  
  
 **if** choice == 1:  
 length = float(input(**"Enter length of rectangle: "**))  
 width = float(input(**"Enter width of rectangle: "**))  
 rect\_area = area(length, width)  
 rect\_perimeter = perimeter(length, width)  
 print(**"Area:"**, rect\_area, **"Perimeter:"**, rect\_perimeter)  
  
 **elif** choice == 2:  
 radius = float(input(**"Enter radius of circle: "**))  
 circle\_area = area(radius)  
 circle\_perimeter = circumference(radius)  
 print(**"Area:"**, circle\_area, **"Perimeter:"**, circle\_perimeter)  
  
 **else**:  
 print(**"Invalid choice"**)  
  
main()

Let’s test run this program. We try circle first.

This program calculates area and perimeter of a rectangle or circle.

Enter 1 for rectangle or 2 for circle: 2

Enter radius of circle: 10

Area: 314.15999999999997 Perimeter: 62.832

The test run is successful. Area and perimeter are calculated and displayed correctly.

Next, let’s try rectangle.

This program calculates area and perimeter of a rectangle or circle.

Enter 1 for rectangle or 2 for circle: 1

Enter length of rectangle: 10

Enter width of rectangle: 5

Traceback (most recent call last):

File "C:/Users/mleun\_000/Desktop/geometryProject/geometry2.py", line 27, in <module>

main()

File "C:/Users/mleun\_000/Desktop/geometryProject/geometry2.py", line 14, in main

rect\_area = area(length, width)

TypeError: area() takes 1 positional argument but 2 were given

We got an error message when we try to call the area function to calculate area of the rectangle. The error message says one argument is expected but two were given. The area function of the rectangle module expects two arguments. Why did we get this error message?

The problem is a name clash. Both the rectangle and circle modules have an area function. When we call the area function, the computer actually executes the area function of the circle module even though we intend to execute the area function of the rectangle module.

How do we resolve this name clash if we still want to use the from-import statement to import functions from modules? Python allows use to rename a function when it is imported to avoid name clash:

**from** rectangle **import** area **as** areaRec, perimeter  
**from** circle **import** area **as** areaCir, circumference

The first statement renames the area function of the rectangle module to areaRec, while the second statement renames the area function of the circle module to areaCir. The following is the modified code:

**from** rectangle **import** area **as** areaRec, perimeter  
**from** circle **import** area **as** areaCir, circumference  
  
**def** main():  
  
 *""" Calculates area and perimeter of rectangle or circle """* print(**"This program calculates area and perimeter of a rectangle or circle."**)  
 choice = int(input(**"Enter 1 for rectangle or 2 for circle: "**))  
  
 **if** choice == 1:  
 length = float(input(**"Enter length of rectangle: "**))  
 width = float(input(**"Enter width of rectangle: "**))  
 rect\_area = areaRec(length, width)  
 rect\_perimeter = perimeter(length, width)  
 print(**"Area:"**, rect\_area, **"Perimeter:"**, rect\_perimeter)  
  
 **elif** choice == 2:  
 radius = float(input(**"Enter radius of circle: "**))  
 circle\_area = areaCir(radius)  
 circle\_perimeter = circumference(radius)  
 print(**"Area:"**, circle\_area, **"Perimeter:"**, circle\_perimeter)  
  
 **else**:  
 print(**"Invalid choice"**)  
  
main()

Notice that we use the name areaRec when we call the area function of the rectangle module, and the name areaCir when we call the area function of the circle module. The following is a test run:

This program calculates area and perimeter of a rectangle or circle.

Enter 1 for rectangle or 2 for circle: 2

Enter radius of circle: 10

Area: 314.15999999999997 Perimeter: 62.832

The following is another test run:

This program calculates area and perimeter of a rectangle or circle.

Enter 1 for rectangle or 2 for circle: 1

Enter length of rectangle: 10

Enter width of rectangle: 5

Area: 50.0 Perimeter: 30.0

Both of these test runs are successful.

# **12.8 Further Readings**

Please read chapter 7 of the textbook. Section 7.1 and 7.2 demonstrate how to use top-down design to develop and code a program. Section 7.3 introduces Python modules and namespaces. Section 7.4 describes the calendar year program in details.