ICS3U: Creating Modules

Recall our earlier function `get_integer`, that prompts the user to enter an integral value with possible `high` and `low` limits.

```
import math

def get_integer(low=-math.inf, high=math.inf, prompt="Enter a value: "):
   integer = int(input(prompt))
   while integer < low or integer > high:
      integer = int(input("Invalid value, try again: "))
   return integer
```

Let's create another function, `get_pos_integer`, that does something similar but uses a lower limit of one. Note how it simply calls our already-written function `get_integer`, so that we do not have to rewrite the entire function from scratch.

```
def get_pos_integer(high=math.inf, prompt="Enter a value: "):
   integer = get_integer(low=1, high=high, prompt=prompt)
   return integer
```

These are such common actions that we might want to use them in *all* of our programs. We could cut-and-paste them in to each program, but this would be a nightmare to maintain: if we wanted to make a change or add a feature to a function, we would have to update the code in every program that uses it. It would be far better if we could save these functions in one file, and have all other programs access them directly from that file. That way, changing the code across all programs would be as simple as changing the code in that one file.

Python allows us to do this by saving the functions in a **module**. We have already used some Python modules in our programs: `math` and `random`. A module is a regular file that contains Python code – there is no need to do anything special to the file to make it a module. To access the functions inside of it, we can `import` it and use the name of the module as a prefix for each function call. For example, let's save the two functions above in a file called `integers.py`, and in a new file, write the following.

```
import integers

num = integers.get_integer(low=1, high=5, prompt="Pick a value between 1 and 5: ")
print("You picked", num)
```

Just like pre-installed modules, prefixing the function name with `integers` tells the Python interpreter that the function is located in a file called `integers.py`. This file should be in the same directory (folder) as our actual program file. There are ways to `import` modules from other directories as well, but this is beyond the scope of this course, so for now, just remember to save your module alongside your actual program.

```
directory
  ├─ main_program.py
  └─ module.py
```

Here is another example where we have written a function, `find_roots`, that determines the real solutions to a quadratic equation in the form $ax^2+bx+c=0$. A second function, `print_roots`, is also included to display the roots based on their number: zero, one or two. Note the use of `None` in the functions. `None` is a reserved value in Python that represents nothing at all. We could have rewritten the functions to use another value, such as the string "none", to achieve the same result. Since we have not yet discussed tuples or lists in any meaningful way, we must return two values each time the function is called, even if there is only one (or zero) real root.

```
import math

# Determine up to two real roots of a quadratic using the Quadratic Formula
def find_roots(a, b, c):
   discriminant = b**2 - 4 * a * c
   if discriminant < 0:
      return None, None
   elif discriminant == 0:
      root = -b / (2 * a)
      return root, root
   else:
      root1 = (-b + math.sqrt(discriminant)) / (2 * a)
      root2 = (-b - math.sqrt(discriminant)) / (2 * a)
      return root1, root2

# Display the real roots, based on their number.
def print_roots(a, b, c):
   root1, root2 = find_roots(a, b, c)
   if root1 == None:
      print("There are no real roots.")
   elif root1 == root2:
      print("There is one real root at ", root1, ".", sep="")
   else:
      print("There are two real roots at ", root1, " and ", root2, ".", sep="")
```

Save this code to a file called `roots.py`. Save the main program below to a separate file called `root_finder.py`.

```
import roots

print("This program finds the roots of a quadratic equation ax^2+bx+c=0.")
a = int(input("Enter a value for a: "))
b = int(input("Enter a value for b: "))
c = int(input("Enter a value for c: "))
roots.print_roots(a, b, c)
```

Try running the main program several times, with different values for a, b and c. See if you can determine values for which there are two, one and zero real roots. Any program that needs to determine these values (such as one that you might write as part of a physics or mathematics course) can utilize these functions by importing them as necessary.

There are some additional things that we can do with modules that may be useful. Note that each time we want to call a function that is inside of a module, we must prefix it with the name of the module, as in `roots.print_roots`. If we want to use one or more specific functions from a module without this requirement, we can use an alternate `import` notation using the `from` keyword.

```
from roots import print_roots

print("This program finds the roots of a quadratic equation ax^2+bx+c=0")
a = int(input("Enter a value for a: "))
b = int(input("Enter a value for b: "))
c = int(input("Enter a value for c: "))
print_roots(a, b, c)
```

This gives the main program access to the `print_roots` function directly. Note that the last line of the main program does not use the `roots` prefix. To import multiple functions from a module, separate them with commas.

If we want to obtain access to *all* functions within a module using this method, it is possible to write the following.

```
from roots import *
```

In this case, `*` is a wildcard that represents everything in the module. Doing this will remove the need to prefix any function with the name of the module. There is a downside to this: if two modules, each containing a function (or value) with the same name are imported, then this will cause an issue with the Python interpreter. When the function is called, which function should be executed? This is not as uncommon as you might think. As programs grow larger and are split between team members, it is more likely that two programmers will develop similar functions with the same names. For this reason, it is usually a good idea to avoid using this method unless you are certain that this will not happen.

Finally, it is also possible to change the prefix applied to imported functions by assigning an **alias**. This is done using the `as` keyword. Below is an example of using an alias to shorten a module's name.

```
import probability_functions as prob
```

Rather than using `probability_functions.my_function`, one can write `prob.my_function` instead. This can be useful if the module has a very long name, or if it has a common abbreviation; however, for what you gain in convenience, you lose in readability. A programmer that is familiar with Python might not recognize `r.choice` whereas `random.choice` is more obvious.