```
(*
                          CS51 Lab 7
                  Modules and Abstract Data Types
 *)

(* Objective: This lab practices concepts of modules, including files
as modules, signatures, and polymorphic abstract data types.

There are 4 total parts to this lab. Please refer to the following
files to complete all exercises:

-> lab7_part1.ml -- Part 1: Implementing modules (this file)
   lab7_part2.ml -- Part 2: Files as modules
   lab7_part3.ml -- Part 3: Interfaces as abstraction barriers
   lab7_part4.ml -- Part 4: Polymorphic abstract types
 *)

(*======================================================================
Part 1: Implementing Modules

*Modules* are a way to package together and encapsulate types and values
(including functions) into a single discrete unit.

By applying a *signature* to a module, we guarantee that the module
implements at least the values and functions defined within it. The
module may also implement more as well, for internal use, but only those
specified in the signature will be exposed and available outside the
module definition. This form of abstraction, information hiding,
implements the edict of compartmentalization.

Below is a `MATH` signature; we'll use it to describe a limited subset
of functions and values that a mathematics module might contain.
......................................................................*)

module type MATH =
  sig
    (* the constant pi *)
    val pi : float
    (* cosine of an angle in radians *)
    val cos : float -> float
    (* sine of an angle in radians *)
    val sin : float -> float
    (* sum of two numbers *)
    val sum : float -> float -> float
    (* maximum value in a list; None if list is empty *)
    val max_opt : float list -> float option
  end ;;

(*......................................................................
Exercise 1A: Complete the implementation of a module called `Math`
that satisfies the signature above. Feel free to make use of various
functions in the `Stdlib` module
<https://caml.inria.fr/pub/docs/manual-ocaml/libref/Stdlib.html>. *)

(* (You may wonder, what's that `nan` in our dummy definition below? The
value `nan` stands for "not a number" and is an actual value of the
`float` type, as dictated by the IEEE Floating Point standard described
at <https://en.wikipedia.org/wiki/IEEE_754>. We're using it here as a
temporary value pending your putting in appropriate ones.) *)
(*......................................................................*)

module Math : MATH =
  struct
    let pi = nan
    let cos _ = nan
```

```
    let sin _ = nan
    let sum _ _ = nan
    let max_opt _ = None
  end ;;

(*..................................................................
Exercise 1B: Now that you've implemented the `Math` module, use it to
compute the maximum of the cosine of pi and the sine of pi, a value of
type float option. Name the resulting value `result`. (Use explicit
module prefixes for this exercise, not global or local opens.)
..................................................................*)

let result = Some nan ;;

(*..................................................................
Exercise 1C: Reimplement the computation from 1B above, now as
`result_local_open`, but using a local open to write your computation
in a more succinct manner.
..................................................................*)

let result_local_open = Some nan ;;
```