OCaml Standard Library

OCaml v. 3.12.0 — June 7, 2011 — — **http://www.ocamlpro.com/**

Standard Modules

Basic Data Types

`Pervasives`	All basic functions
`String`	Functions on Strings
`Array`	Functions on Polymorphic Arrays
`List`	Functions on Polymorphic Lists
`Char`	Functions on Characters
`Int32`	Functions on 32 bits Integers
`Int64`	Functions on 64 bits Integers
`Nativeint`	Functions on Native Integers

Advanced Data Types

`Buffer`	Automatically resizable strings
`Complex`	Complex Numbers
`Digest`	MD5 Checksums
`Hashtbl`	Polymorphic Hash Tables
`Queue`	Polymorphic FIFO
`Stack`	Polymorphic LIFO
`Stream`	Polymorphic Streams
`Map`	Dictionaries (functor)
`Set`	Sets (functor)

System

`Arg`	Argument Parsing
`Filename`	Functions on Filenames
`Format`	Pretty-Printing
`Genlex`	Simple OCaml-Like Lexer
`Marshal`	Serialization Functions
`Lexing`	Functions for `ocamllex`
`Parsing`	Functions for `ocamlyacc`
`Printexc`	Generic Exception Printer
`Random`	Random Number Generator
`Printf`	printf-like Functions
`Scanf`	scanf-like Functions
`Sys`	OS Low-level Functions

Tweaking

`Lazy`	Functions on Lazy Values
`Gc`	Garbage Collection Tuning
`Weak`	Weak Pointers (GC)

Popular Functions per Module

module Hashtbl

```
let t = Hashtbl.create 117
Hashtbl.add t key value;
let value = Hashtbl.find t key
Hashtbl.iter (fun key value -> ... ) t;
let cond = Hashtbl.mem t key
Hashtbl.remove t key;
Hashtbl.clear t;
```

module List

```
let len = List.length l
List.iter (fun ele -> ... ) l;
let l' = List.map(fun ele -> ... ) l
let l' = List.rev l1
let acc' = List.fold_left (fun acc ele -> ...) acc l
let acc' = List.fold_right (fun ele acc -> ...) l acc
if List.mem ele l then ...
if List.for_all (fun ele -> ele >= 0) l then ...
if List.exists (fun ele -> ele < 0) l then ...
let neg = List.find (fun x -> x < 0) ints
let negs = List.find_all (fun x -> x < 0) ints
let (negs,pos) = List.partition (fun x -> x < 0) ints
let ele = List.nth 2 list
let head = List.hd list
let tail = List.tl list
let value = List.assoc key assocs
if List.mem_assoc key assocs then ...
let assocs = List.combine keys values
let (keys, values) = List.split assocs
let l' = List.sort compare l
let l = List.append l1 l2 or l1 @ l2
let list = List.concat list_of_lists
```

Functions using Physical Equality in List

`memq`, `assq`, `mem_assq`

Non-tail Recursive Functions in List

`append`, `concat`, `@`, `map`, `fold_right`, `map2`, `fold_right2`, `remove_assoc`, `remove_assq`, `split`, `combine`,`merge`

module String

```
let s = String.create len
let s = String.make len char
let len = String.length s
let char = s.[pos]
s.[pos] <- char;
let concat = prefix ^ suffix
let s' = String.sub s pos len'
let s = String.concat "," list_of_strings
let pos = String.index_from s pos char_to_find
let pos = String.rindex_from s pos char_to_find
String.blit src src_pos dst dst_pos len;
let s' = String.copy s
let s' = String.uppercase s
let s' = String.lowercase s
let s' = String.escaped s
String.iter (fun c -> ...) s;
if String.contains s char then ...
```

module Array

```
let t = Array.create len v
let t = Array.init len (fun pos -> v_at_pos)
let v = t.(pos)
t.(pos) <- v;
let len = Array.length t
let t' = Array.sub t pos len
let t = Array.of_list list
let list = Array.to_list t
Array.iter (fun v -> ... ) t;
Array.iteri (fun pos v -> ... ) t;
let t' = Array.map (fun v -> ... ) t
let t' = Array.mapi (fun pos v -> ... ) t
let concat = Array.append prefix suffix
Array.sort compare t;
```

module Char

```
let ascii_65 = Char.code 'A'
let char_A = Char.chr 65
let c' = Char.lowercase c
let c' = Char.uppercase c
let s = Char.escaped c
```

module Buffer

```
let b = Buffer.create 10_000
Printf.bprintf b "Hello %s\n" name
Buffer.add_string b s;
Buffer.add_char b '\n';
let s = Buffer.contents s
```

module Digest

```
let md5sum = Digest.string str
let md5sum = Digest.substring str pos len
let md5sum = Digest.file filename
let md5sum = Digest.channel ic len
let hexa = Digest.to_hex md5sum
```

module Filename

```
if Filename.check_suffix name ".c" then ...
let file = Filename.chop_suffix name ".c"
let file = Filename.basename name
let dir = Filename.dirname name
let name = Filename.concat dir file
if Filename.is_relative file then ...
let file = Filename.temp_file prefix suffix
let file = Filename.temp_file ~temp_dir:"." pref suf
```

module Marshal

```
let string = Marshal.to_string v
               [Marshal.No_sharing; Marshal.Closures]
let _ = Marshal.to_buffer string at_pos max_len v []
let (v : of_type) = Marshal.from_string string at_pos
if String.length s > at_pos + Marshal.header_size then
   let needed = Marshal.total_size s at_pos in ...
```

module Random

```
Random.self_init ();
Random.init int_seed;
let int_0_99 = Random.int 100
let coin = Random.bool ()
let float = Random.float 1_000.
```

module Printexc

```
let s = Printexc.to_string exn
let s = Printexc.get_backtrace ()
Printexc.register_printer (function
     MyExn s -> Some (Printf.sprintf ...)
   | _ -> None);
```

module Lazy

```
let lazy_v = lazy (f x)
let f_x = Lazy.force lazy_v
```

module Gc

```
Gc.compact ();
Gc.major ();
Gc.set { Gc.get() with
   Gc.minor_heap_size = 1_000_000;
   Gc.max_overhead = 1_000_000; (* no compaction *)
 };
```

module Weak

```
let t = Weak.create size
Weak.set t pos (Some v);
match Weak.get t pos with None -> ...
```

module Arg

```
let arg_list = [
  "-do", Arg.Unit (fun () -> ..), ": call with unit"
; "-n", Arg.Int (fun int -> ..), "<n> : call with int"
; "-s", Arg.String (fun s -> ..), "<s> : call/w string"
; "-yes", Arg.Set flag_ref, ": set ref"
; "-no", Arg.Clear flag_ref, ": clear ref" ]
let arg_usage = "prog [args] anons: run prog with args"
Arg.parse arg_list (fun anon -> .. ) arg_usage;
Arg.usage arg_list arg_usage;
```

module Map

```
module Dict = Map.Make(String)
module Dict = Map.Make(struct
    type t = String.t let compare = String.compare end)
let empty = Dict.empty
let dict = Dict.add "x" value_x empty
if Dict.mem "x" dict then ...
let value_x = Dict.find "x" dict
let new_dict = Dict.remove "x" dict
Dict.iter (fun key value -> ..) dict;
let new_dict = Dict.map (fun value_x -> ..) dict
let nee_dict = Dict.mapi (fun key value -> ..) dict
let acc = Dict.fold (fun key value acc -> ..) dict acc
if Dict.equal dict other_dict then ...
```

module Set

```
module S = Set.Make(String)
module S = Set.Make(struct
    type t = String.t let compare = String.compare end)
let empty = S.empty
let set = S.add "x" empty
if S.mem "x" set then ...
let new_set = S.remove "x" set
S.iter (fun key -> ..) dict;
let union = S.union set1 set2
let intersection = S.inter set1 set2
let difference = S.diff set1 set2
let min = S.min_elt set
let max = S.max_elt set
```