OCaml Standard Library

Standard Modules

- `Stdlib`: All basic functions
- Basic data types: `Array`, `Bool`, `Bytes`, `Char`, `Float`, `Fun`, `Int`, `Int32`, `Int64`, `List`, `Nativeint`, `Option`, `Result`, `String`, `Unit`
- Advanced data types: `Bigarray`, `Buffer`, `Complex`, `Digest`, `Hashtbl`, `Lazy`, `Map`, `Queue`, `Seq`, `Set`, `Stack`, `Stream`, `Uchar`
- System: `Arg`, `Filename`, `Format`, `Genlex`, `Lexing`, `Marshal`, `Parsing`, `Printexc`, `Printf`, `Random`, `Scanf`, `Sys`
- Tweaking: `Callback`, `Ephemeron`, `Gc`, `Oo`, `Weak`

Popular Functions per Module

module List

```
let l = List.init 10 (fun i -> i)
let len = List.length l
let acc' = List.fold_left (fun acc ele -> ...) acc l
let acc' = List.fold_right (fun ele acc -> ...) l acc
List.iter (fun ele -> ... ) l;
List.iteri (fun index ele -> ... ) l;
let l' = List.map(fun ele -> ... ) l
let l' = List.mapi(fun index ele -> ... ) l
let l' = List.filter_map(fun ele -> ... ) l
let l' = List.rev l1
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 x = List.find (fun x -> x < 0) ints
let x_o = List.find_opt (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 list 2
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 String.compare l
let l = List.append l1 l2 or l1 @ l2
let list = List.concat list_of_lists
```

- Functions using physical equality: `memq`, `assq`, `mem_assq`
- Non-tail recursive functions: `append`, `concat`, `@`, `map`, `mapi`, `fold_right`, `map2`, `fold_right2`, `remove_assoc`, `remove_assq`, `split`, `combine`, `merge`
- Raising **`Not_found`**: `find`, `assoc`, `assq`.
- Raising **`Failure`**: `hd`, `tl`, `nth`.
- Raising **`Invalid_argument`**: `nth`, `nth_opt`, `init`, `iter2`, `map2`, `rev_map2`, `fold_left2`, `fold_right2`, `for_all2`, `exists2`, `combine`

module Hashtbl

```
let t = Hashtbl.create ~random:true 117
Hashtbl.add t key value;
Hashtbl.replace t key value;
let value = Hashtbl.find t key (* Not_found *)
let value_o = Hashtbl.find_opt t key
Hashtbl.iter (fun key value -> ... ) t;
let cond = Hashtbl.mem t key
Hashtbl.remove t key;
Hashtbl.clear t;
```

module String

```
let s = String.make len char
let len = String.length s
let char = s.[pos]
let concat = prefix ^ suffix
let s' = String.sub s pos len'
let s = String.concat "," list_of_strings
let p' = String.index_from s p char_to_find
let p' = String.rindex_from s p char_to_find
String.blit src src_pos dst_bytes dst_pos len;
let s' = String.uppercase_ascii s
let s' = String.lowercase_ascii s
let s' = String.escaped s
String.iter (fun c -> ...) s;
if String.contains s char then ...
let l = String.split_on_char ',' s
assert ("abc" = String.trim "  abc   ");
```

- Deprecated: `set`, `create`, `copy`, `fill`, `uppercase`, `lowercase`, `capitalize`, `uncapitalize`
- Raising **`Invalid_argument`**: `get`, `set`, `create`, `make`, `init`, `sub`, `fill`, `concat`, `escaped`, `index_from`, `index_from_opt`, `rindex_from`, `rindex_from_opt`, `contains_from`, `rcontains_from`
- Raising **`Not_found`**: `index`, `rindex`, `index_from`, `rindex_from`

module Bytes

```
let b = Bytes.create length
let b' = Bytes.make length char
let b' = Bytes.init length (fun i -> ...)
let b' = Bytes.copy b
let b' = Bytes.extend b left right
Bytes.blit src srcoff dst dstoff len;
let b = Bytes.concat sep blist
Bytes.iteri (fun i c -> ...) b;
let s = Bytes.unsafe_to_string b
let b = Bytes.unsafe_of_string s
let i = Bytes.get_uint8 b index
Bytes.set_int32_le b pos 0l;
(* Bytes.[sg]et_u?int(8|(16|32|64)_[lbn]e) *)
```

module Array

```
let t = Array.make 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;
```

modules Int, Int32, Int64, Nativeint

```
module I = Int (* / Int32 / Int64 / Nativeint *)
let x = I.add I.zero I.one
let y = I.mul x (I.succ x)
let d,r = I.div y x, I.rem y x
let x' = I.abs (I.neg I.minus_one)
let z = I.shift_left (I.logor x y) 2
let z' = I.shift_right z 2
let z' = I.shift_right_logical z 2
(* unsigned operations not in Int *)
let c : int = I.unsigned_compare I.max_int I.min_int
let du, ru = I.unsigned_div y x, I.unsigned_rem y x
```

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 (* Not_found *)
let value_x_o = Dict.find_opt "x" dict
let new_dict = Dict.remove "x" dict
let dict' = Dict.update "x"
          (function None -> ... | Some v -> ...) 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 String.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 -> ..) set;
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
```

module Char

```
let ascii_65 = Char.code 'A'
let char_A = Char.chr 65
let c' = Char.lowercase_ascii c
let c' = Char.uppercase_ascii 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';
Buffer.add_utf_8_uchar b uc;
let s = Buffer.contents b
```

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_o = Filename.chop_suffix_opt ~suffix:".c" name
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 Seq

```
let s = List.to_seq l (* works with most containers *)
let s = Array.to_seqi a
(* lazy functions *)
let s = Seq.map (fun e -> e) s
let s = Seq.filter (fun e -> true) s
let s = Seq.filter_map (fun e -> Some e) s
let s = Seq.flat_map (fun e -> Seq.return e) s
(* immediate functions *)
let acc = Seq.fold_left (fun acc e -> ...) acc s
Seq.iter (fun () -> ...) s;
(* getting one value, (recalculating) *)
let open Seq in
  match s () with
  | Nil -> None
  | Cons (e,s_tl) -> Some e
```

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

```
(* $OCAMLRUNPARAM=b *)
let s = Printexc.to_string exn
let s = Printexc.get_backtrace ()
Printexc.register_printer (function
     MyExn s -> Some (Printf.sprintf ...)
   | _ -> None);
Printexc.set_uncaught_exception_handler
  (fun e raw_b -> () );
try ... with e ->
   let b = Printexc.get_raw_backtrace () in
   ...
   Printexc.raise_with_backtrace e b
```

module Ephemeron

```
module E1 = Ephemeron.K1
let e = E1.create ()
E1.set_key e k;
E1.set_data e d;
E1.blit_key e_from e_to;
if E1.check_data e then ...
module EHASH = E1.Make ( Hashable )
```

module Lazy

```
let lazy_v = lazy (f x)
let f_x = Lazy.force lazy_v
let f_x = match lazy_v with lazy f_x -> f_x
```

module Arg

```
let arg_list = [
  "-do", Arg.Unit (fun () -> ..), ": call with unit";
 "-n", Arg.Int (fun int -> ..), "<n> : with int";
 "-s", Arg.String (fun s -> ..), "<s> : with 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.parse_dynamic
    (ref arg_list) (fun anon -> ...) arg_usage;
Arg.usage arg_list arg_usage;
let arg_list = Arg.align arg_list
```

module Printf

```
Printf.printf "flush\n%!";
let s = Printf.sprintf "%s=%d or %x\n" string int hexa
Printf.fprintf oc "Error: %dL=%dl\n" int64 int32;
Printf.bprintf buf "%.3f if %b" float boolean;
Printf.printf "%a" (fun oc x -> ...) x;
```

module Format

Formatters:

`@[<h> ...@]`	Horizontal box: everything on one line
`@[<v> ...@]`	Vertical box: next line at every break hint
`@[<hv> ...@]`	Switch from horizontal box to vertical box if needed
`@[<b> ...@]`	Indented box
`@[<hov> ...@]`	Fill line after line
`@[<... 5> ... @]`	Next line in box indented by 5
`@␣`	Breakable space
`@,`	Break hint
`@;`	Full break
`@?`	Flush
`@<4>%i`	Print int on 4 chars
`@.`	Close everything and flush

module Bigarray

```
module B1 = Bigarray.Array1
let a = B1.create Bigarray.char Bigarray.c_layout length
let e = B1.get a i
B1.set a i e;
B1.blit
  (B1.sub_left src src_ofs len)
  (B1.sub_left dst dst_ofs len);
```