Factor is a modern stack-based language, based on Forth, created by Slava Pestov.

Code in this file can be typed into Factor, but not directly imported because the vocabulary and import header would make the beginning thoroughly confusing.

```
! This is a comment

! Like Forth, all programming is done by manipulating the stack.
! Stating a literal value pushes it onto the stack.
5 2 3 56 76 23 65    ! No output, but stack is printed out in interactive mode

! Those numbers get added to the stack, from left to right.
! .s prints out the stack non-destructively.
.s     ! 5 2 3 56 76 23 65

! Arithmetic works by manipulating data on the stack.
5 4 +    ! No output

! `.` pops the top result from the stack and prints it.
.    ! 9

! More examples of arithmetic:
6 7 * .        ! 42
1360 23 - .    ! 1337
12 12 / .      ! 1
13 2 mod .     ! 1

99 neg .       ! -99
-99 abs .      ! 99
52 23 max .    ! 52
52 23 min .    ! 23

! A number of words are provided to manipulate the stack, collectively known as shuffle words.

3 dup -          ! duplicate the top item (1st now equals 2nd): 3 - 3
2 5 swap /       ! swap the top with the second element:        5 / 2
4 0 drop 2 /     ! remove the top item (dont print to screen):  4 / 2
1 2 3 nip .s     ! remove the second item (similar to drop):    1 3
1 2 clear .s     ! wipe out the entire stack
1 2 3 4 over .s  ! duplicate the second item to the top: 1 2 3 4 3
1 2 3 4 2 pick .s ! duplicate the third item to the top: 1 2 3 4 2 3

! Creating Words
! The `:` word sets Factor into compile mode until it sees the `;` word.
: square ( n -- n ) dup * ;    ! No output
5 square .                     ! 25

! We can view what a word does too.
! \ suppresses evaluation of a word and pushes its identifier on the stack instead.
\ square see    ! : square ( n -- n ) dup * ;

! After the name of the word to create, the declaration between brackets gives the stack effect.
! We can use whatever names we like inside the declaration:
: weirdsquare ( camel -- llama ) dup * ;
```

```
! Provided their count matches the word's stack effect:
: doubledup ( a -- b ) dup dup ; ! Error: Stack effect declaration is wrong
: doubledup ( a -- a a a ) dup dup ; ! Ok
: weirddoubledup ( i -- am a fish ) dup dup ; ! Also Ok

! Where Factor differs from Forth is in the use of quotations.
! A quotation is a block of code that is pushed on the stack as a value.
! [ starts quotation mode; ] ends it.
[ 2 + ]       ! Quotation that adds 2 is left on the stack
4 swap call . ! 6

! And thus, higher order words. TONS of higher order words.
2 3 [ 2 + ] dip .s      ! Pop top stack value, run quotation, push it back: 4 3
3 4 [ + ] keep .s       ! Copy top stack value, run quotation, push the copy: 7 4
1 [ 2 + ] [ 3 + ] bi .s ! Run each quotation on the top value, push both results: 3 4
4 3 1 [ + ] [ + ] bi .s ! Quotations in a bi can pull values from deeper on the stack: 4 5 ( 1+3 1+4 )
1 2 [ 2 + ] bi@ .s      ! Run the quotation on first and second values
2 [ + ] curry       ! Inject the given value at the start of the quotation: [ 2 + ] is left on the stack

! Conditionals
! Any value is true except the built-in value f.
! A built-in value t does exist, but its use isn't essential.
! Conditionals are higher order words as with the combinators above.

5 [ "Five is true" . ] when                    ! Five is true
0 [ "Zero is true" . ] when                    ! Zero is true
f [ "F is true" . ] when                       ! No output
f [ "F is false" . ] unless                    ! F is false
2 [ "Two is true" . ] [ "Two is false" . ] if   ! Two is true

! By default the conditionals consume the value under test, but starred variants
! leave it alone if it's true:

5 [ . ] when*      ! 5
f [ . ] when*      ! No output, empty stack, f is consumed because it's false


! Loops
! You've guessed it.. these are higher order words too.

5 [ . ] each-integer               ! 0 1 2 3 4
4 3 2 1 0 5 [ + . ] each-integer   ! 0 2 4 6 8
5 [ "Hello" . ] times              ! Hello Hello Hello Hello Hello

! Here's a list:
{ 2 4 6 8 }                        ! Goes on the stack as one item

! Loop through the list:
{ 2 4 6 8 } [ 1 + . ] each          ! Prints 3 5 7 9
{ 2 4 6 8 } [ 1 + ] map             ! Leaves { 3 5 7 9 } on stack

! Loop reducing or building lists:
{ 1 2 3 4 5 } [ 2 mod 0 = ] filter  ! Keeps only list members for which quotation yields true: { 2 4 }
{ 2 4 6 8 } 0 [ + ] reduce .       ! Like "fold" in functional languages: prints 20 (0+2+4+6+8)
```

```
{ 2 4 6 8 } 0 [ + ] accumulate . .  ! Like reduce but keeps the intermediate values in a list: prints { 0 2 6 12 }
1 5 [ 2 * dup ] replicate .       ! Loops the quotation 5 times and collects the results in a list: { 2 4 8 16 32
1 [ dup 100 < ] [ 2 * dup ] produce ! Loops the second quotation until the first returns false and collects the

! If all else fails, a general purpose while loop:
1 [ dup 10 < ] [ "Hello" . 1 + ] while  ! Prints "Hello" 10 times
                                        ! Yes, it's hard to read
                                        ! That's what all those variant loops are for

! Variables
! Usually Factor programs are expected to keep all data on the stack.
! Using named variables makes refactoring harder (and it's called Factor for a reason)
! Global variables, if you must:

SYMBOL: name            ! Creates name as an identifying word
"Bob" name set-global   ! No output
name get-global .       ! "Bob"

! Named local variables are considered an extension but are available
! In a quotation..
[| m n                  ! Quotation captures top two stack values into m and n
 | m n + ]              ! Read them

! Or in a word..
:: lword ( -- )           ! Note double colon to invoke lexical variable extension
   2 :> c                 ! Declares immutable variable c to hold 2
   c . ;                  ! Print it out

! In a word declared this way,  the input side of the stack declaration
! becomes meaningful and gives the variable names stack values are captured into
:: double ( a -- result ) a 2 * ;

! Variables are declared mutable by ending their name with a shriek
:: mword2 ( a! -- x y )   ! Capture top of stack in mutable variable a
   a                      ! Push a
   a 2 * a!               ! Multiply a by 2 and store result back in a
   a ;                    ! Push new value of a
5 mword2                  ! Stack: 5 10

! Lists and Sequences
! We saw above how to push a list onto the stack

0 { 1 2 3 4 } nth         ! Access a particular member of a list: 1
10 { 1 2 3 4 } nth        ! Error: sequence index out of bounds
1 { 1 2 3 4 } ?nth        ! Same as nth if index is in bounds: 2
10 { 1 2 3 4 } ?nth       ! No error if out of bounds: f

{ "at" "the" "beginning" } "Append" prefix    ! { "Append" "at" "the" "beginning" }
{ "Append" "at" "the" } "end" suffix          ! { "Append" "at" "the" "end" }
"in" 1 { "Insert" "the" "middle" } insert-nth ! { "Insert" "in" "the" "middle" }
"Concat" "enate" append                       ! "Concatenate" - strings are sequences too
"Concatenate" "Reverse " prepend              ! "Reverse Concatenate"
{ "Concatenate " "seq " "of " "seqs" } concat ! "Concatenate seq of seqs"
{ "Connect" "subseqs" "with" "separators" } " " join  ! "Connect subseqs with separators"
```

```
! And if you want to get meta, quotations are sequences and can be dismantled..
0 [ 2 + ] nth                           ! 2
1 [ 2 + ] nth                           ! +
[ 2 + ] \ - suffix                      ! Quotation [ 2 + - ]
```

Ready For More?

- Factor Documentation