

## *Dos and Don'ts*

Perl values spring into existence!

```
# The long way...
if (exists $browsers{$string}) {
    $browsers{$string}++;
} else {
    $browsers{$string} = 1;
}
# The Perl way..
$browsers{$string}++;
```

# Modularity: Functions and Packages

- When programs gets larger, it gets difficult to see its structure.
- Sometimes identical actions have to be executed for different inputs:

```
if(runCalculation 'HF') {  
    runCalculation 'LDA';  
}
```

- Functions provide means to achieve that!
- Packages allow to group several functions of similar purpose.

# Function Declaration

Arguments are passed via @\_ array.

```
sub myFunction {
    print "Hello, World!\n";
    print 'I was called with ' . scalar @_ . " args.\n";
}
#...
myFunction 12, 3, 4,4; # 4 arguments;
@days = ('mon', 'tue', 'wed');
myFunction 'sun', @days; # 4 arguments, too.
```

# Example Function Usage

```
#!/usr/bin/perl
```

```
=head1 My test program
```

```
Extract from FILE lines containing given REGEX.
```

```
=cut
```

```
sub extractData {  
    my ($string, $fname) = @_;  
    open FL, $fname or  
        die "Cannot open $fname for data extraction\n";  
    @rows = grep /$string/, <FL>;  
    close FL or warn "Cannot close $fname\n";  
    return @rows;  
}
```

```
die "Call me REGEX FILE\n" unless 2 == @ARGV;  
print extractData @ARGV;
```

# Prototypes

Prototypes help to avoid mistakes in the number and type of arguments.

```
sub myGrep {
  print "My Argument list: ". (join ':', @_) ."\n";
}
myGrep 'a', 'b'; # My Argument List: a:b.
#
sub yourGrep($) {
  print "My Argument list: ". (join ':', @_) ."\n";
}
yourGrep 'a', 'b'; # Useless use of a constant in a void context.
```

## Passing Hashes and Arrays

Perl collapses all arguments into one list:

```
sub myFunc {  
    print "My Argument list: ". (join ':', @_) . "\n";  
}  
@a = (1,2,3); @b = (4,5,6);  
myFunc @a, @b; # 1:2:3:4:5:6
```

They have to be passed as so-called references to keep them separated:

```
sub myFunc {  
    print 'I have ' . @_ . " arguments.\n";  
    print "Elements:". join(', ', @{$_[0]}), "\n";  
}  
@a = (1,2,3); @b = (4,5,6);  
myFunc \@a, \@b; # I have 2 elements: 1,2,3
```

# Returning Values

- Returning scalar values and simple list is trivial.
- Separate lists should be returned via references.

```
sub myAnimals { ...; return $noOfDogs; }  
my $dogCount = myAnimals();  
#...  
sub fishAndDogs { return (\@fish, \@dogs); }  
my ($myFish, $myDogs) = fishAndDogs();  
my @fish = @$myFish;
```

# Using Packages

- Collect larger routines in packages so that they can be reused.
- Easy to use: `use XXX;`

Example:

```
use List::Util;  
print List::Util::sum(1..3), "\n";  
# import sum into main namespace:  
use List::Util qw(sum);  
print sum(1..3), "\n";
```

# Package Definitions

Create a separate file `Gaussian.pm` with following content:

```
package Gaussian;
sub run {
  ...
}
1; # This is to inform the including program that
    # initialization succeeded.
```

- Use it in the usual way with  
use Gaussian; ... Gaussian::run 'input.log';
- Perl consults the special array `@INC` for directories that contain package files.

## Further Reading

- Chapter 6 describes functions in details.

### The Last Assignment:

Use functions to split orthogonal functionality, for readability and maintainability.