LECTURE 10: FLOW CONTROL & SUBROUTINES

FLOW CONTROL

- More options than Python
- Conditional statements: if, unless
- Loop statements
  - while
  - until
  - for
  - foreach
- Modifiers

CONDITIONAL STATEMENTS

- If statement controls the following block
  - if, elsif, else
- unless is opposite of if
  - Equivalent to if(not $Boolean)
  - unless, elsif, else
- See flow-control-if-unless

LOOP STATEMENTS: WHILE, UNTIL, DO

- while: Loops while boolean is true
- until: Loops until boolean is true
  - Opposite of while
- do: At least one loop, then depends on while or until

```
while ($hungry) {
    $hungry = eat($banana);
}
do {
    $answer = get_answer();
    $correct = check($answer);
} until ($correct);# } while (!$correct)
```

LOOP STATEMENTS: FOR, FOREACH

- for: Like C: for(initialization; condition; increment)
- foreach: Iterates over a list or array
- Good to localize loop variables with my
- See flow-control-for-foreach

MODIFIERS

- Simple statements can take single modifier
  - Places emphasis on the statement, not the control
  - Can make programs more legible
  - Parentheses usually not needed
  - Good for setting default values
  - Valid modifiers are if, unless, while, until, foreach
  - See modifier-pl

BASIC PRINTING

```
print "Hello world.";
print 102;
print "102";
print 10+2;
print "10+2";
```

BASIC PRINTING

```
print "1. And then he said, 'How are you?'\n";
print '2. And I said, \'Fine, fine.\'\n';
print '3. And I said, "Fine, fine."\n';
print "4. And I said, \'Fine, fine.\'\n";
print "5. Look at my fine backslash:\\\n ";
```

Output:

```
1. And then he said, 'How are you?'
2. And I said, 'Fine, fine.'\n3. And I said, "Fine, fine."\n4. And I said, 'Fine, fine.'
5. Look at my fine backslash:\
```

PRINT WITH Q AND QQ

```
$first= q<That's how you make a single quote>;
$second= qq{I said, "This is how you make a double quote"};
print $first;
print "\n\n";
print $second;
print "\n\n";
```

Output:

```
That's how you make a single quote

I said, "This is how you make a double quote"
```

PRINTING VARIABLES

```
$a='Apple';
$b='Jacks';
print "The best cereal in the world is $a $b.\n";
print 'The best cereal in the world is $a $b.\n';
```

Output:

```
The best cereal in the world is Apple Jacks.
The best cereal in the world is $a $b.\n
```

PRINTING META CHARACTERS

```
print "I have $15.00 in my pocket.\n";
print 'I have $15.00 in my pocket.’;
print "I have \$15.00 in my pocket.\n";

print "My email address is james@james.com.\n";
print "My email address is james\@james.com.\n";

$color="black";
$number="eight";
print "\n\nPlease find my favorite color and number below:\n\n";
print "\tMy favorite color:\t\t\t$color\n";
print "\tMy favorite number is:\t\t\t$number\n\n";
```

A meta character is a character that has a special meaning (instead of a literal meaning) to a computer program.

CHOP() VS CHOMP()

- **chop** removes the last character of the string completely, and returns the removed character
- **chomp** only removes the last character if it is a newline character, and returns the total number of characters removed from its arguments

EXAMPLE CODE

```
$str ="Look, it's a lion";
$a=chop($str);
print $a . " " . $str . "\n";

$str ="Look, it's a lion\n";
$a=chop($str);
print $a . " " . $str . "\n";
```

```
$str ="Look, it's a lion";
$a=chomp($str);
print $a . " " . $str . "\n";

$str ="Look, it's a lion\n";
$a=chomp($str);
print $a . " " . $str . "\n";
```

SUBROUTINES

- Subs group related statements into a single task
- Perl allows various ways of handling arguments
- Perl allows various ways of calling subs
- Perl also supports anonymous subs

DECLARING SUBROUTINES

- Subroutines are declared with the sub keyword
- Subroutines return values
  - Explicitly with the return command
  - Implicitly as the value of the last executed statement
- Return values can be a scalar or a flat list
  - wantarray describes what context was used
  - Unused values are just lost

```
sub ten { return wantarray() ? (1 .. 10) : 10; }
@ten = ten(); # (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
$ten = ten(); #10
($ten)= ten(); #(1)
($one, $two)= ten(); # (1, 2)
```

HANDLING ARGUMENTS

- Pass by value, pass by reference
- Arguments are passed into the **@_** array
  - Usually should copy **@_** into local variables

```
sub add_one {#Pass by value example
my ($n) = @_; # Copy first argument
return($n+1); #Return 1 more than argument in a list
sub plus_plus { #Pass by reference example
$_[0] = $_[0] + 1; } # Modify first argument
my ($a, $b) = (10, 0);
add_one($a); # Return value is lost, nothing changes $a still 10
$b = add_one($a); # $a: 10, $b: 11, scalar context takes last list
element

plus_plus($a); # Return value lost, but a now is 11,$a is 11 now
$b = plus_plus($a); #$a and $b are both 12 now}}
```

CALLING SUBROUTINES

- Arguments in parentheses
  - Parentheses are not needed if sub is declared first. But using parentheses is often good style
- Subroutines are another data type

```
sub factorial {
    my ($n) = @_;
    return $n if $n<= 2;
    $n * factorial($n - 1);
}
```

HANDLING ARGUMENTS

- Pass by value, pass by reference
- Arguments are passed into the **@_** array
  - Usually should copy **@_** into local variables

```
sub add_one {#Pass by value example
my ($n) = @_; # Copy first argument
return($n+1);
} #Return 1 more than argument in a list
sub plus_plus { #Pass by reference example
$_[0] = $_[0] + 1; } # Modify first argument
my ($a, $b) = (10, 0);
add_one($a); # Return value is lost, nothing changes $a still 10
$b = add_one($a); # $a: 10, $b: 11, scalar context takes last list
element

plus_plus($a); # Return value lost, but a now is 11,$a is 11 now
$b = plus_plus($a); #$a and $b are both 12 now
```

CALLING SUBROUTINES

- Arguments in parentheses
  - Parentheses are not needed if sub is declared first. But using parentheses is often good style
- Subroutines are another data type

```
print factorial(5).”\n” # parentheses required

sub factorial {
   my ($n) = @_;
   return $n if $n<= 2;
   $n * factorial($n - 1);
}
print factorial 5 # parentheses not required
```

DIE

```
sub fibonacci { # declare subroutines
   my ($n) = @_; # copy arguments
   die "Number must be positive" if $n<= 0;# check arguments
   return 1 if $n<= 2; # perform computation
   return (fibonacci($n-1) + fibonacci($n-2)); # return results
}

foreach my $i (1..5) {
   my $fib = fibonacci($i);
   print "fibonacci($i) is $fib\n";
}
```

REFERENCES

- References indirectly refer to other data
- Dereferencing yields the data
- References allow to create anonymous data
- References allow to build hierarchical data structures
- See references-anonymous-hierarchy-data

REFERENCING DATA

- References indirectly refer to other data
  - References are like pointers
  - Backslash operator creates a reference
  - References are scalars

```
my @fruit = qw(apple banana cherry);
my $fruitref = \@fruit;
```

DEREFERENCING DATA

- Dereferencing yields the data
  - Appropriate symbol dereferences original data
  - Arrow operator(->) dereferences items

```
my @fruit = qw(apple banana cherry);
my $fruitref = \@fruit;
print "I have these fruits:@$fruitref.\n";
print "I want a $fruitref->[1].\n";
```

ANONYMOUS DATA

- References allow to create anonymous data
- Build unnamed arrays with brackets ([]) and a ref.
- Build unnamed hashes with braces ({}) and a ref.

```
my $fruits = ["apple", "bananas", "cherries"];
my $wheels = {unicycle =>1, bike =>2, car =>4};
print “I like $fruits->[0]\n”;
print "A car has $wheels->{car}wheels.\n";
```

HIERARCHICAL DATA

- References allow you to build hierarchical data structures
- Arrays and hashes can only contain scalars
- A reference is a scalar, even if it refers to an array or a hash
- May omit the arrow operator for these structures

```
my $fruits = ["apple", "bananas", "cherries"];
my $veggies = ["spinach", "turnips"];
my $grains = ["rice", "corn"];
my @shopping_list = ($fruits, $veggies, $grains);
print "I should remember to get $shopping_list[2]->[1].\n";
print "I should remember to get $shopping_list[0][2].\n";
```

FIN!