LECTURE 9: HASHES, PERL OPERATORS AND FLOW CONTROL

LAST LECTURE

- Data types:
  - Scalar ($)
  - Array (@)
  - Hash (%)

HASHES

- Store any number of unordered scalars organized in key/value pairs
- Have a name preceded by an % character
- May be declared in a local scope with my identifier
- Indexed by key name
- May be assigned in several ways
- Dynamically assume whatever values or size needed
- May be sliced
- Easy to iterate

HASH DECLARATION

- Store any number of unordered scalars
  - Numbers, strings, references
  - Any combination of above
- Organized in key/value pairs
- Sample definition:

```
my %wheels = (unicycle =>1, bike =>2, tricycle =>3,car =>4,
semi =>18);
```

HASH DECLARATION

- Indexed by key name
- See hash-defn-index-modify

```
print "A bicycle has $wheels{bike} wheels.\n";
$wheels{bike} = 4; # Adds training wheels
print "A bicycle with training wheels has $wheels{bike} wheels.\n";
```

HASH DECLARATION

```
my %dessert = ("pie", "apple", "cake", "carrot", "sorbet", "orange");

%dessert = (pie =>"apple", cake =>“carrot”, sorbet =>“orange”);
%dessert = qw(pie apple cake carrot sorbet orange); # all the same

my %ice_cream = (bowl =>"chocolate", float =>"root beer");

my %choices = (%dessert, %ice_cream);

print "I would like $choices{sorbet} sorbet.\n";
```

See hash-defn2-print

DYNAMIC VALUE OR SIZE AS NEEDED

- Can dynamically lengthen or shorten hashes
- May be defined , but empty
- Old keys may be deleted (with the delete keyword)
- See hash-add-delete
- Take a hash slice with @hash{@keys}

SLICING EXAMPLE

```
my %sounds=(cow => "moooo” , duck => "quack” ,
horse => "whinny” ,sheep => "baa", hen =>"cluck", pig =>"oink");
my @barnyard_sounds = @sounds{"horse", "hen", "pig"};
print "I heard the following in the barnyard:
@barnyard_sounds\n";
```

ITERATION EXAMPLE

- keys returns a list of the keys
- values returns a list of the values
- each returns key/value pairs in random order
- while loop can iterate over entire hash
- See: hash-slicing-iterating

OPERATORS

- Perl has many types of operators
  - Numeric
  - String
  - Boolean
  - List
- perldoc perlop gives complete descriptions

NUMERIC OPERATORS

- All common operators are provided
  - Increment and decrement (++, --)
  - Arithmetic (+, -, *, /, %, **)
  - Assignment (=, +=, -=)
  - Bitwise (<<, >>)

STRING OPERATORS

- All common operators are provided
- Common operators are provided
  - Concatenation (.)
  - Assignment (.=, x=)
  - Repetition (x)
  - See operators-string-quoting-boolean

WHAT'S NEXT

- Operators/operations continued
- Flow control
  - conditionals
  - loops
- Subroutines

QUOTING OPERATIONS

- Quoting operator constructs strings
- Many types of quoting are provided

```
my $cat = "meow";
my $sound = "$cat"; # $sound = "meow"
my $variable = '$cat'; # $variable = "$cat"
print "$variable says $sound\n";
$sound = qq{"meow"};
$sound = qq("meow");
print "$variable says $sound\n";
```

BOOLEAN OPERATIONS

- Many types of operators are provided
  - Relational (<,>, lt, gt);
  - Equality (==, !=, eq, ne)
  - Logical (high precedence) (&&, ||, !)
  - Logical (low precedence) (and, or, not)
  - Conditional (?:)

```
my ($x, $y) = (12, 100);
my $smaller = $x<$y ? $x : $y;
print "The smaller number is $smaller.\n";
The smaller number is 12.
```

NUMERIC AND STRING COMPARISONS

- Separate operators for numeric vs. string comparison
  - Numeric (<=,>=)
  - String (le, ge)

```
my ($a, $b) = ("apple", "orange");
print "1: apples are oranges\n" if ($a eq $b); # False
print "2: apples are oranges\n" if ($a == $b); # True!
my ($x, $y) = (12, 100);
print "3: $x is more than $y\n" if ($x gt $y); # True!
print "4: $x is more than $y\n" if ($x>$y); # False
```

LIST OPERATORS AND FUNCTIONS

- Many useful operations or functions are built-in
  - sort
  - reverse
  - push/pop
  - unshift/shift
  - split/join
  - grep
  - map
- See operators-list

SORT AND REVERSE

- sort: Sorts the list, alphabetically by default
- reverse:
  - In scalar context, concatenates the list and reverses the resulting string
  - In list context, reverses the list. With hash, swap key and value per pair

```
my $output = reverse ("dog", "fish", "horse");
my @animals = qw(dog cat fish parrot hamster);
my @sorted = reverse sort @animals;
print "I have the following pets: @sorted\n";
my $word = "backwards";
my $mirror = reverse $word;
print qq("$word" reversed is "$mirror"\n);
%by_address = reverse %by_name;
```

SPLIT AND JOIN

- split: Splits a string into a list of substrings
  - Removes a delimiting string or regular expression match
- join: Joints a list of substrings into a single string
  - Adds a delimiting string

```
my @animals = qw(dog cat fish parrot hamster);
my $string = join(" and a ", @animals);
print "I have a $string.\n";
my $sentence = "The quick brown fox...";
my @words = split(" ", $sentence);
```

LIST OPERATORS: GREP

- Similar to the unix command grep
- Finds matching items in the list
- Matches usually based on a regular expression or a comparison

```
my @juices = qw(apple cranapple orange grape apple-cider);
my @apple = grep(/apple/, @juices);
print "These juices contain apple: @apple\n";

my @primes = (2, 3, 5, 7, 11, 13, 17, 19);
my @small = grep {$_<10} @primes;# $_ is each element of @primes
print "The primes smaller than 10 are: @small\n";
```

LIST OPERATORS: MAP

- Maps an input list to an output list
- Powerful, but mapping can be complex
- $_ is the "fill in the blank" scalar

```
my @primes = (2, 3, 5, 7, 11, 13, 17, 19);
my @doubles = map {$_ * 2} @primes;
print "The doubles of the primes are: @doubles\n";

# grep {$_<10} @primes
my @small = map {$_<10 ? $_ : ()} @primes;
print "The primes smaller than 10 are: @small\n";
```

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); }
```

FIN!