Variables and Data Structures

Aims

- Understand the different data types
- Manipulate numbers and strings with operators
- Tell the difference between scalar, array and hash

Data

- Functions generally need something to act upon - Data
- This can either be
  - Use once data – numbers and strings
  - Keep it around data - variables
- We will use some different functions than print/say. Please ask if I don't mention what the are trying to do, but we'll look more at some of them later

Numbers

- bin/01-integer.pl

  say 1;
- Integers don't need quoting

  say 0;
- 0 is just a number (well, actually its a concept...)

Maths

say 1+1; # addition

*say 2*7; # multiplication*

say 9-3; # subtraction

say 9/3; # division

- Your standard four maths operations

Maths

*say 3**2; # square*

*say 4**4; # (4 to the power of 4)*

- ** gives you to the power of

A string operation

say 1 . 1;

- We'll look at it when we do strings, but we have just concatenated these together

Boolean numbers

say 1 if 0; # this is boolean untrue

say 4 if 4; # any number other than 0 is true

- 0 is untrue, all other numbers are true (including negatives)

Floating Point

say 1.00; # truncates past the decimal point

say '1.00'; # acts as a string

say 1.01;

- Perl truncates trailing (and leading) 0's unless specified as a string

Producing Sequences

```
say 0..9;
```

- The double dot operator doesn't concatenate twice, but instead allows a sequence of consecutive numbers to be produced quickly

Numerical Testing

```
say 4 if ( 1 == 1 ); # lhv and rhv are equal
say 5 if ( 1 != 2 ); # lhv and rhv are not equal
say 6 if ( 2 > 1 ); # lhv greater than rhv
say 7 if ( 1 < 2 ); # rhv less than lhv
```

- = is an assignment operator in perl (and most languages) so we use == to check equivalence of numbers
- *also >= and =<*

Comparisons

```
say 10 <=> 11; # -1
say 10 <=> 10; # 0
say 11 <=> 10; # 1
```

- the comparison or 'spaceship' operator
- 3 possible results are given

  -1 if the right hand side is greater than left, 0 if the same,

  1 if the left hand side is greater.
- Useful if you want to sort a set of numbers.

Strings

- bin/02-string.pl

  ```
  say 'Hello World';
  ```
- Strings need to be quoted,

  ```
  say “Hello World”;
  ```
- Either single or double quotes

Length

```
say length 'Hello World';
```

- The length function gives us the length of the string in characters
- *You should note here, Perl is working right to left. 'Hello World' is passed to length, the output of this (11) is passed to say.*

Manipulating Strings

```
say lc 'TACtgATTACACATTACT'; # lower case the string
say uc 'tactgattacACATTact'; # upper case the string
say lcfirst 'TACTGATTACACATTACT'; # lower case the first letter
say ucfirst 'tactgattacacattact'; # upper case the first letter
```

Manipulating Strings

- User inputs name ANdrEw

```
say ucfirst lc 'ANdrEw';  # result Andrew
```

- So you can do some sanity checking!

Concatenate Strings

```
say 'Hello' . 'World'; # concatenate
```

- We saw concatenating two numbers previously. Now we are using strings.

Different Whitespace Characters

```
say "Hello\tWorld,\nHow are you today?"; # contains a newline and and tab character
```

- For different whitespace characters (tab/newline..) we can use standard character representations, but you must use double quotes.

Boolean Strings

```
say ''; # empty string, this is boolean false
say 'true string' if 'true'; # any string is true apart from the empty string
say 'true empty string' if ''; # this won't be shown, as empty string is false
```

- An empty string is false, everything else is true, incl space. (Uses ASCII value to denote truthfulness)

Producing Sequences

```
say 'a'..'z'; # quick alphabet
```

- Exactly the same as with numbers, you can use the double dot to generate a list.
- However, be careful if you meant to concatenate 'Hello'.'World'

```
say 'Hello'..'World;
```

Maths(?)

```
say 'a' * 2;
say 'a' + 'b';
```

- Not numeric warnings – ends up 0

```
say 'a' x 2;
```

- What you probably meant to do! Repeat a 2 times
- Note: In Perl number operators act on numbers, and Perl makes all non-number characters 0.

String Testing

```
say 'a equals a' if ( 'a' eq 'a' );
say 'a equals b' if ( 'a' eq 'b' );
say 'a does not equal b' if ( 'a' ne 'b' );
say 'a less than b' if ( 'a' lt 'b' );
say 'b greater than a' if ( 'b' gt 'a' );
```

- Because number operators only work on numbers, we need a string equivalent

String Comparisons

```
say 'a' cmp 'b'; # -1
say 'a' cmp 'a'; # 0
say 'b' cmp 'a'; # 1
```

- The same is true with comparison operators

```
say 'a' <=> 'b';
```

- not numeric warning, ends up 0 - equivalent

Number/String Comparison Table

	Number	String
Equal	==	*eq*
Not Equal	!=	*ne*
Less Than	<	*lt*
Gr Than	>	*gt*
Compare	<=>	*cmp*

Substrings

```
say substr 'TACTGATTACACATTACT', 0, 4;
say substr 'TACTGATTACACATTACT', 5, 12;
say substr 'TACTGATTACACATTACT', 5, 15;
say substr 'TACTGATTACACATTACT', -5, 3;
```

- As you'll see later, we use 0-based counting
- A negative offset counts from the end
- If you leave off the length, then you get all remaining characters

String Location

```
say index 'TACTGATTACACATT', 'GATTACA';
say index 'TACTGATCACATTACT', 'TACT', 7;
```

- We can get the location within a string of a character set
- Adding an offset skips over that many bases before searching

A math warning!

```
say 20/'b';
```

- This will break your code!
- Strings equate to 0, so this produces a divide by zero warning!

Other Things With Strings

- There are loads of other things to do with strings, such as regular expressions.
- We will come to these in time, but lets move onto holding strings and numbers around for longer - Variables.

Variables

- There are 3 basic types of Variable
  - Scalar – holds 1 piece of data
  - Array – holds a 'list' of pieces of data
  - Hash – holds a 'list' of named pieces of data
- Variables enable you to pass information around and operate on the data contained
- **Everything** you do in a program is likely to need to use a variable of some type

Scalar

- The simplest variable is a scalar
- A named 'box' containing one piece of data
- That data can be anything (more on that later)

bin/03-scalar.pl

$scalar

- Scalars are always denoted by a $ sign
  - Think of a special S as part of the name
- The first character of the name must be a letter
- The rest of the name can be
  - Letters
  - Numbers
  - Underscore

Naming

- Choose a name which represents what the variable should contain

  $sequence

  $amino_acid

  $height
- Not just some arbitrary names

  $one

  $foo

Assignment

- Use the = operator

```
my $integer_scalar = 10;
my $string_scalar = 'I am a string';
say $integer_scalar;
say $string_scalar;
```

Typing

- Perl uses dynamic typing, so this is OK

```
my $integer_scalar = 'I am a string';
```

- But DO NOT do it. Anyone looking at the code will be confused.

What can I do with it?

- Use a scalar exactly as though the number or string were there

```
say $integer_scalar + $integer_scalar;
say $string_scalar . $integer_scalar;
say $string_scalar * $integer_scalar;
```

What can I do with it?

```
say $string_scalar * $integer_scalar;
```

- This gave a warning. You probably wanted

```
say $string_scalar x $integer_scalar;
```

- x is a 'repeat' operator

Mix fixed and variable

- We can mix fixed data and variables

```
say 'I can use a fixed string and concatenate to ' .
$string_scalar;

say 30 / $integer_scalar;
```

String interpolation

```
say 'Hello, ' . $string_scalar;
```

- We don't have to concatenate these together

```
say 'Hello, $string_scalar'; # wrong
```

- Use double quotes

```
say "Hello, $string_scalar"; # right
```

- Double quotes interpolate variables, as wel as doing the right thing with \t,\n...

undef

- There is a 'magic' value **undef**
- All variables are assigned this if nothing else was assigned
- It is boolean **false**

```
my $undef_scalar;
say $undef_scalar
```

undef

- You can explicity assign undef

```
$string_scalar = undef;
say $string_scalar if $string_scalar;
```

- Only do this if you need to clear something out (say to recapture some memory after dumping the entire Lord of the Rings, with Appendices and The Hobbit to a scalar)

Arrays

- Arrays are a named group of scalars
- They are like a list
- Read top to bottom, or jumped to by an index
- Each position in the array is a scalar box, which is acted upon in exactly the same way as a scalar

bin/04-array.pl

Creating an Array

- Arrays are named with exactly the same rules as a scalar, except
  - They begin with an @ symbol
  - Usual to be a plural word
- They can be generated in a number of ways, but we still use the = assignment operator.

Creating an Array

```
my @string_array = qw{hello i am an array of strings};
```

- qw{} quotes each word, and we assign the quoted list to an array

```
my @integer_array = 1..10;
```

- As we saw before, 1..10 generates a sequence of integers, which we can assign to

Creating an Array

```
my @comma_made_array = ( 'i', 'see', 'a', 'little',
'silhoutte', 'of', 'a', 'man,' );
```

- Possibly the most common. Note the (), this tells perl to treat as an array of items

```
my @x_generated_array = ( 'scaramouche,' ) x 2;
```

- We can use the repeat operator to repeat an item multiple times, and then assign to the array

Array Operations

```
say 'The string_array, without any join: ' .
@string_array;
```

- Output directly concatenates all elements together.

```
say 'The integer_array, with a comma join: ' . join ',',
@integer_array;
```

- We can join them with a separator

Array Operations

```
say 'The fourth number in the integer array is ' . $integer_array[3];
```

- To obtain an individual element, change the @ to a $ (scalar), and then add the index with the [x] notation

```
say 'The last word in the string array is ' . $string_array[-1];
```

- Negative indices count from the end

Array Operations

```
my @slice = @string_array[1 .. 4];
say 'An array slice of the string array, indices 1->4: ' . join ' ', @slice;
```

- A subset of the array (slice) can be obtained by assigning to an array, with a sequence for the index

Array Operations

```
say 'The number of elements in string_array is : ' . scalar @string_array;
```

- length is string length, scalar is array length

```
say 'The length of x_generated_array is : ' . length @x_generated_array;
```

- The keyword scalar gives you the number of elements in the array. In some cases you can omit the word, but if you are always explicit, then you won't have a potential bugsite!

Interpolation

```
say 'The first word in the string array is $string_array[0]';
say "The first word in the string array is $string_array[0]";
```

- Just as we did with a scalar, we can insert into "" the scalar of an element in the array, and it will be interpolated into the string.

Doing things with Arrays

```
say join ' ', @comma_made_array, @x_generated_array, 'will you do the fandango?';
```

- Functions tend to expect arrays of arguments (we'll explain more later) but as such, we can use the comma to 'join' arrays together, so that the function treats as one array.

Doing things with Arrays

```
my @three_lines = ( ( join ' ', @comma_made_array ) , ( join ' ', @x_generated_array ), 'will you do the ballroom blitz?' );
say scalar @three_lines;
say join "\n", @three_lines;
```

- We can make new arrays from data processed

Iterating over Arrays

To iterate over an array, we can use the function foreach and act with each. Perl is basically taking each element in turn, assigning it's value to a scalar ($int/$str) and then you use this as the variable within the block of code (we'll look at code blocks later as well).

Iterating over Arrays

```
foreach my $int ( @integer_array ) {
  say $string_array[$int];
}
foreach my $str ( @string_array ) {
  say $integer_array[$str];
}
```

- Each element is assigned to a scalar, which we then use in the block of code

Empty Arrays

```
my @assigned_into_array;
```

- No need to explicitly do '= ()'

- Empty arrays are boolean false, as there are 0 elements.
- If there any elements the array is boolean true (incl. if they themselves are false)

Adding, Replacing and Removing Elements

```
$assigned_into_array[2] = 10;
```

- We can assign directly to an index. If there is already a value at that index, you will overwrite it.

```
say scalar @assigned_into_array;
```

- indices that are missing are autovivified, with *undef* as the value

```
say @assigned_into_array;
```

Adding, Replacing and Removing Elements

```
say scalar @comma_made_array;
push @comma_made_array, @x_generated_array;
say scalar @comma_made_array;
say join ' ', @comma_made_array;
```

- push adds elements to the end of an array. That can be a single element, or another array.

Adding, Replacing and Removing Elements

```
my $last_element = pop @comma_made_array;
say scalar @comma_made_array;
say join ' ', @comma_made_array;
```

- pop removes the last element of the array. Note it completely removes it, including the 'box' that it was in, so that the number of elements is also changed.

Adding, Replacing and Removing Elements

```
my $first_element = shift @comma_made_array;
say scalar @comma_made_array;
say join ' ', @comma_made_array;
```

- shift removes the first element of the array. Again, completely, so it will alter the length of the array, and also the other index positions

Adding, Replacing and Removing Elements

```
unshift @comma_made_array, $last_element;
say scalar @comma_made_array;
say join ' ', @comma_made_array;
```

- unshift puts elements onto the start of the array. It can be a single element or and array, and they will remain in that order.
- The indices of other elements will change

Adding, Replacing and Removing Elements

```
splice @comma_made_array, 3, 0, @x_generated_array;
say scalar @comma_made_array;
say join ' ', @comma_made_array;
```

- splice allows you to muck around with the internals of the array. This example adds one array into another, starting at index 3
- There are more examples in bin/04-array.pl

Hashes

- Unfortunately, a problem occurs with arrays which make it difficult to store lots of data. You need to remember the order the data is in

```
my @personal_info = ( qw{Andy Brown 180 brown blue} );
```

- I could have dyed my hair blue
- Depending on the country, Andy could be my family name

Hashes

- Fortunately, we have a solution – associative arrays, or hashes.

```
my %personal_info = (
  forename => 'Andy',
  surname  => 'Brown',
  height   => '180',
  hair     => 'brown',
  eyes     => 'blue',
);
```

Hashes

- We can then access the data with the name of the data (key)

```
say $personal_info{forename};
```

- *bin/05-hash.pl*

Creating a Hash

- Hashes are named with exactly the same rules as scalars, except
  - They begin with an % symbol
  - Plurality tends to depend on the information
- They can be generated in a number of ways, but we still use the = assignment operator.

2 Methods of Assignment

- You can create in two ways, 1 clearer than the other
- Unclear:

```
my %string_key_and_value_hash = qw{key1 val1 key2 val2 key3 val3};
```

- It is constructed in a similar way to an array, so could be confusing

```
# my %string_key_and_value_hash = ('key1', 'val1', 'key2', 'val2', 'key3', 'val3');
```

2 Methods of Assignment

- Clearer:

```
my %string_key_any_value_hash = (
  key1 => 'val1',
  key2 => 'valB',
  key3 => '3',
  key4 => undef,
);
```

- => acts like a comma (the fat comma)
- Always use this form!

Outputting data

```
say %string_key_any_value_hash;
```

- The output of this is in a random order, which is due to the internal way perl stores the data

```
foreach my $key ( sort keys %string_key_any_value_hash ) {
  say $key . ' : ' . $string_key_any_value_hash{$key};
}
```

- keys %x gives an array of the keys only, which we sort and then use in foreach

Interpolation

```
say 'The first value in the string_key_and_value_hash is $string_key_and_value_hash{key1}';
say "The first value in the string_key_and_value_hash is $string_key_and_value_hash{key1}"; # again, double quotes interpolate the variable
```

- As before, we can interpolate the values from a hash using double quotes directly into a string

Keys can be Numerical

```
my %keys_are_numbers = (
  1 => 'one', 2 => 'two', 3 => 'three',
);
say $keys_are_numbers{2};
```

- There are no real benefits to this, as arrays are faster, except possibly
  - You don't have to start at 0
  - You won't autovivify the missing indices if you want to assign to a later number

Summary

- Strings and Numbers can be processed, by various functions and operations
- Scalars are the basic 'storage' unit in a perl program, and contain 1 string or number
- Arrays are lists of elements (scalars) that are ordered by index
- Hashes (associative arrays) contain scalars which are named and accessed with a key