Scripting Languages

SHELL, PERL, RUBY, AND PYTHON… OH BOY!

Shell

SH, CSH, BASH, KSH, ZSH
AND
A TALE OF TWO UNIXES

The Bourne Shell

- Once operating systems were created, you needed some way of talking to them
- Far harder than just… talking to an operator…
- Unix v7 brought us the Bourne shell… written by Stephen Bourne.
- Replaced the Thompson shell, which was written by Ken Thompson, and was horrible.
- Widely used for all your operating system, file redirection, execution needs.

The C Shell

- Csh, or the C Shell, was the next big and popular shell
- Much like everything else in Linux/Unix land, there was a great war between sh users and csh users.
- Csh was different to sh because csh used C syntax.
- As C was the dominant language in Unix, it was rather intuitive for the Unix user
- Csh was also the default shell of the Berkeley Software Distribution.

The Korn Shell

- The Korn shell (ksh) is another shell from the mid 1980s, written by David Korn for Bell Labs (the people who owned non-BSD Unix at the time)
- Essentially, the C compatibility of the csh was awesome, and the Old Unixers wanted some of it
- But they wouldn't *use* BSD programs, because of the strange nature of Unix programmers
- So instead, ksh was built to be compatible with sh code, as well as implementing the most fun aspects of csh.

The Tenex-like C Shell

- Tcsh (tee-cee-shell), was another 1980s improvement on shell languages from the other side of the pond, so to speak.
- TENEX, a now long dead operating system, had an interesting shell language that allowed tab-file-completion and contextual history.
- So, tcsh was built as a compatible-with-csh upgrade for csh.
- Tcsh is the current default shell used by most children of BSD, with the exception of Darwin/OSX, which uses bash.

The Bourne Again Shell

- Bash is the shell language you are most comfortable with.
- You've been using it the whole time you've been here at Curtin at the very least!
- Was built as a GNU replacement of sh for the GNU/Hurd project in 1989.
- Instead became the primary os language of all *nix types.
- Linux has used bash since the very beginning.
- Is similar to ksh in capabilities, although can run in a restricted mode that emulates regular sh.

The Z Shell

- Zsh is the “newest” member of the shell family.
- Created in 1990 as an extention of the bash project
- Contains all the capabilities of bash, as well as many of the capabilities of tcsh.
- Can emulate sh, csh, tcsh, bash….
- Runs into similar problems as Perl due to there being too many ways to do the same thing.
- Has strange differences to csh and sh that can confuse new users.

That's a lotta shells

- Shell scripts are designed for specialised purposes
- Not general programming like say, ALGOL, ADA, C, etc.
- This is reflected in both the syntax and functionality of the languages
- Shell programs are never compiled, all interpreted at run time.
- Need to ensure that your script is in the command path, and to ensure that the file is executable.
- Far slower than compiled programs.

Shell Syntax

- Very simple, and a lot less functionality
- Case sensitive(!) and some keywords are just regular expression special characters.
- However, to aid readability, scripts do have:
  - Actual keywords
  - While not blocks, control structures do have starts and ends.
- Oh also, there are no activation records either.
- The interpreters handle everything in situ.

Shell Data Types

- Don't exist.
- Well to be honest they do, but they are only weakly typed
- No data types:
  - C shell has an array, but the array starts at ***1***
  - Ksh has the Compound Variable, which is basically a variant record if you remember those from Pascal.
- Local variables:
  - Given variables that are known only to the shell that created them. There are no Activation Records. Data is lost to child shells
- Environment variables:
  - Are available to the shell or child shells that are spawned.
  - This must be done explicitly with the export keyword.
- Variables can be unset, unless they are read only. Some variables are special and provided by the language.

Shell Functions

- Are used to execute a series of commands with a name
  - Kinda like mini-scripts, once defined they are part of a shell's memory, so it does not need to read from disk when called.
- Function names are a name followed by ().
  - Function definitions are a set of commands separated by ; and enclosed in {}
- Function runs in current environment
  - Shares all variables with current script, allows user to pass arguments too.
- Functions only exist in your current shell. They *cannot* be exported to sub shells.

Shell Control Structures

- Evaluation of conditional statements: TEST
- IF, FOR, WHILE, CASE, UNTIL exist, but don't use block structure consistently.
- Sh provides a way to break out of loops or return to the top, violates structured programming
  - Shift: shifts parameter list to the left a number of times
  - Break: exits loop
  - Continue: returns to start of loop

More Shell Control Structures

- Some control structures work differently or unexpectedly
  - For, for example, works on a list and implicitly works through it. In procedural languages, it's based on incrementing some variable. This strange loop is sometimes called For-Each.
- Numeric processing may or may not be done in the shell. Sh has no ability to do math, must use expr, bc, or awk.
  - Csh, tsch, zsh all can though, but can *also* use these.

More Shell Control Structures

- Basic exception handling on par with C. Trapping signals from the kernel.
- Regular expressions are often built in, and make shell languages very good at pattern recognition
- Can also allow inter process communication via "pipes" (C and other languages can do this too, but it's far easier in shell land)
- Input redirection has explicit commands, can redirect input from console or file with ease.
- Basically, it's made for managing *other programs*.

PERL

COMPUTING EPISODE III: REVENGE OF PL/1

The History of Perl

- In 1987, Larry Wall was trying to make a reporting script for a Unix machine.
- It needed to read log files and tell him in a succinct way what was going on.
- However, the shell languages of the time were not helpful enough…
- So, Larry made his own. Perl. An acronym with no meaning.
- Early Perl was actually ok… it's just that nobody told Larry when to stop…
- Soon, Perl was enormous. Far larger than even Ada.

Why Perl is important

- Perl is an important lesson in what not to do as a program designer.
- Perl has had every single concept in program language design strapped onto it, with little regard for whether or not it's a good idea.
- As such, it both does and does not support OO.
- It does and does not support functional design.
- It does and does not support many, many things.
- *Don't be like Perl*.

So, Perl

- Is a scripting language. Scripts are both compiled and interpreted, the interpreter real time compiles and executes the code.
- Perl is portable, can work on non *nix systems
- Perl has a motto: "There's More Than One Way To Do It"
  - Combines shell scripting, *and* C.
- Extremely flexible, more than any other language.
- This is why Perl is the lead code golf language.

The issue with Perl

- Rather than having a few commands…. Perl has *every* command, so there are many ways of doing any given thing
- Many constructs, shortcuts, *side effects*…
- The result is that there are many, *many* different *ever so slightly* different ways of doing the same thing, which makes any one person's Perl code almost incomprehensible to another.
- This makes Perl very large, complex, and complicated.

Perl is difficult to work with

- Very easy to write (just write what you know)
- Very hard to maintain (you don't know what other people know)
- Maintenance is entirely predicated on the developers ability to document their code. And make it readable/comprehensible.
- You could write a C-like program, or could write some monstrous series of shortcut commands. Same result.
- Extremely hard to master. Maybe impossible.

Data Types

- Perl is very “flexible” on data types.
- 1 + 1 = 2
  “1” + “1” = “2” (????)
- There are two kinds of variable, List and Scalar
  - Scalar is numbers, strings…
  - Lists are similar to arrays:
    - () Empty list
    - (a, b) a list with 2 elements
    - (a, ‘hello’, b, 5,1.1) a totally valid list with 5 elements.

Data Structures

- Perl has arrays… and associative arrays.
- Can sort and modify arrays (in many ways)
- Perl stores hashes for associative arrays using keys rather than numbers
  - This can make things interesting, but can be more flexible than Pascal type arrays
- You can also nest data structures in Perl
- And just in case you wanted them, pointer style references do exist, and can be used to access any data in Perl.

Syntax

- Mostly like C. *Mostly*.
- Perl introduces a large number of operators, keywords…. But only a few actually mean anything useful.
- Loop syntax in Perl is just like C, but you must have brackets for everything.
- Same for if statements
- Perl has *separate* relational operators for strings and numbers.
- Perl has many assignment operators (+=, -=, %=, **=)
- Perl is also weakly typed with automatic variable declaration… so you can just up and use variables!

Program Structure

- Perl has blocks much like C.
- However it also has labelled blocks that you can use to jump into and out of loops.
  - Like a GOTO, but more confusing
- Perl does have subroutines
- You invoke them as so: &subroutine
- Subroutines don't *list parameters passed to them or even what their type is*

Subroutine Hell

- Most programming languages tell you what parameters you may or may not pass
  - void function(int a, float b);
- Perl does not care for your petty "structure", it mashes all the variables passed to it into one massive super variable and lets the subroutine figure it out.
- Arrays lose their identities in this way, as they are merged.
- The only way to maintain structure is with *pointers*.

AaaaaAAaaaaAAAAaaAAAAAa!

```
Sub example {
   ($ref1, $ref2) = @_;
   @x = @$ref1; @y = @ref2;
   ….
}
@var1 = {1, 2}; @var2 = {3,4,5,6,7,8,9,10};
&example( \@var1, \@var2};
```

It gets worse

- Subroutines can't be nested in Perl. Why? Reasons.
- Perl has built in functions, that may or may not be functions. They may be:
  - Functions that are functions
  - Functions that are actually operators that take one argument pretending to be functions.
- There are no meaningful ways to separate these.

Variable Scoping Nightmares

- Perl implements *both* static and dynamic scoping for variables
  - Why? Reasons!
- Automatically declared variables (ones you just use) are *global* and *persistant*.
- Variables not explicitly declared with the "my" keyword are *also* global, no matter where they are declared!
- To reduce this madness, Perl lets you create packages, which limit the scope of variables.

Dynamic Variable Scoping

- Perl implements dynamic scoping with the "Local" keyword.
- The variable declared masks any previous global variables for the subroutine, *and any subroutine called from that subroutine*.
- *When does local not mean local??*
- Perl has local ("my") and global variables.
  - Basically, global should be masked by "local", and local *should* be masked by "my", though this doesn't always happen.
- Very hard to detect bugs sometimes.

Control Structures?

- Perl has IF, FOR, DO, WHILE, and FOREACH.
- Perl has some loop controls (last, next, redo)
- Perl has no Case/Switch statement. However, a labelled block system can emulate one.
- And Perl also allows labelled loops
- This allows you to goto without using goto.
- Great for nested loops and driving developers mad.

Example:

```
OUTER: while ($again eq 'y') {
        while () {
          print 'Enter a number: ';
          chomp($num = <STDIN>);
          if ($num eq "exit") {
                  last OUTER;
          }
}
```

Packages

- Very important to the functionality of perl
- Actually define proper scopes.
- Normal package is MAIN
- Packages contain
  - Variables
  - Functions
  - File handles!

Package Syntax

- An example is :

```
package mypackage;
$X=1;
```

- This would allow you to access X as $mypackage::X
- In this way you can perform code reuse, similar to ADA

Object Orientation ??

- Yes, Perl "enables" OO
- Class definition is done by "anonymous hashes", which means while it's *kind* of like OO, it's not really.
- Perl requires no formal syntactic interface to a class's methods.
- Methods are created using "new" and destroyed using "DESTROY".
- Perl implements single *and* multiple inheritance
- Technically inheritance is powered by a variable here, there's no "object" per say, just what looks like one.

Ruby

…ON RAILS? GET IT?

What is Ruby?

- Ruby is a scripting language of the early 1990s
- Perl was bad, really bad. So Ruby tries to do the opposite
- Heavily structured, OO based scripting language.
- Totally interpreted, unlike Perl.
- Borrows from Smalltalk and LISP
- Has exception handling.
- Generally pretty popular, especially with web programmers.

Ruby Data Types

- Numbers, strings, symbols, constants, ranges, and special values (true false etc)
- Arrays
  - Indexed by integers
  - Elements can be of different types
  - Can be resized dynamically
- Hashes
  - Key can be any object at all!

Ruby Syntax

- Similar to Perl and other scripting languages.
- Small number of keywords
- Statements are not delimited by semicolon, instead uses whitespace as statement separation.
- Blocks can begin with braces or begin/end
- Provides many assignment operators like Perl
- Automatic variable declaration, also like Perl

Ruby Program Structure

- Organised around classes and methods.
- Program may only contain methods or basic code blocks.
- Number of arguments to program methods may be variable
- Ruby allows methods to be aliased to names (function pointers without the pointers)
- Code blocks visually similar to C or Perl.

Ruby Method example:

```
def methodname( argument1, argument2, ... )
   .....
   return result
end
```

Variables in Ruby

- Type can change as the program executes
- Type allocation is done based on what value is assigned to the variable
- Type can change just by…. Assigning a new type
- Ruby provides an inbuilt type conversion mechanism
- Ruby also determines what the variable scope is automatically

Four Scopes and Several Rubies Ago

- There are four scope classes for variables in Ruby:
  - Local: Only for the construct in which they appear, denoted by a starting lowercase or underscore character
  - Global: Are global, denoted by $
  - Class: Global to all instances of that class, denoted by @@
  - Instance: Only available to one instance of a class, denoted by @

OO in Ruby

- Ruby is actually object oriented, unlike Perl
- Almost everything is an object.
- Single inheritance only, but provides "mixins" which sort of act as pseudo-interfaces
- Three levels of access:
  - Public
  - Private
  - Protected
- Additional naming encapsulation with Modules

Python

NOBODY SUSPECTS THE SPANISH INQUISITION!

What is Python

- Another early 90's scripting language hoping to fix the same problems Ruby sought to.
- Hybrid interpreted, compiles code dynamically down to virtual machine bytecode like Java uses and then interprets that.
- Has two versions, 2 and 3. They are different.
- Heavily based on readability, rather than strong structure.
- Very widely used these days, more so than even Ruby.

Data Types

- Python has three kinds of data type:
  - Variables: numbers strings characters objects etc
  - Numerically iterable constructs: Lists, tuples, etc.
  - Hash iterable constructs: dictionaries etc.
- Has some special values, True, False, None, Empty…

Python Syntax

- Differs between Python2 and Python3.
- Both have similar keywords (if, for, while, etc)
- Both use whitespace to delineate structure
  - Lines end with /n
  - Blocks are determined by shared indentation
  - Extremely easy to read
  - *Highly structured*.
- Performs automatic type detection (though this can be overridden)

Python Program Structure

- In most basic form is simply a series of statements, similar to a shell script.
- Can have function declarations
- Can have class declarations
- Python is fond of using the labelling principle to allow default values for function parameters
- Python scope dictates that class methods must import “self”, as otherwise the class has no access to it’s own variables!
- Python also allows function aliasing, like Ruby

Variables in Python

- Very similar to Ruby
- Type detection is automatic and dynamic.
- Variable scope is determine by the tabulation and by the interpreter itself
- Classes and functions can also be variables
- Several built in variables exist to interact with the state of the interpreter (__name__ for example)

Object orientation in Python

- Python OO is heavily based on C++
- Multiple inheritance abounds
- Objects are loosely connected with their functions
- Also, OO is not enforced, you can just write scripts that have no objects (and no functions!)
- This makes Python fantastically flexible as a language, without reading like Perl.

When is a Language not a Scripting Langage?

- Python 2 has a lot less structure and regularity than Python 3
  - Some keywords in 2 are just functions in 3
  - Bracket usage in 2 is more lax than in 3
- Python 3 is not compatible with 2
- They are not the same language.
- Python 2 is a freeform scripting language with strong programming elements
- Python 3 is an interpreted programming language with scripting elements.
- Some things that work with 2 don't with 3, and vice versa.

Conclusion!

HOORAY

Shell languages

- Have improved dramatically over their time here
- Are designed to allow operators to manage other programs, not be programs themselves
  - They're really slow after all
  - And have less features than current languages.
- Languages like Perl, Ruby, and Python toe that line though, and possibly crossing it.
- The one you prefer is a very personal choice, and has to do with how you like to program.