Slide 1

Administrivia

- (None?)

Slide 2

A Little About Perl — Introduction

- Initially designed in 1987. Wikipedia says not actually an acronym, but others say it's "Practical Extraction and Reporting Language". Good name, whether official or not.
- General-purpose, interpreted, imperative with support for object-oriented programming.
- Draws heavily on shell-script language, `awk`, and `sed`. Focus is on text manipulation.
- Core philosophy is "There's More Than One Way To Do It" (TMTOWTDI).
- Once widely used for scripting, now maybe being supplanted by Python, but there's a lot of legacy code? *Huge* collection of third-party "modules" (Perl equivalent of classes), available via CPAN.
- Also useful for command-line "one-liners".

Slide 3

Perl — Getting Started

- Documentation available via `perldoc`. Start with `perldoc perlintro` (my starting point for this lecture). (Interestingly(?) enough, documentation often bundled with program source. `man perlpod` for more information.)
- Run programs as, e.g., `perl foobar.pl` (typical extension), or use the convention used in shell scripts (first line is `#!` followed by name of program to use to process the rest) and mark executable.

Slide 4

Perl Basics

- Like Python and Scala, no explicit `main` function.
- "Block-structured" syntax, mostly familiar. Whitespace generally not significant; statements end with semicolons; lines starting with # are comments.
- ("Hello, world" example.)
- Good idea to start all programs with

```
use strict;
use warnings;
```

Slide 5

Perl — Data and Variables

- Data can be text strings, integers, or floating-point values. Strings in single or double quotes (difference is similar to how it works in `bash`).
- Variables are not typed (who said “variables don’t have types, data does”?). Need not be declared but arguably should be.
- Variables can be “scalars”, arrays, hashes, or references.

Slide 6

Perl “Scalars”

- Scalars represent single values (text string or number). Referred to as `$foobar`.
- Need not be declared, but can be — `my $foobar` defines local variable. Undeclared variables are global in scope.

Slide 7

Perl — Arrays

- Expandable arrays, containing any kind of data, or a mix.
- Declare with, e.g., `my @array`. Can initialize with list (e.g., `(1, 'hello', 2, 'bye')`).
- Reference individual elements as, e.g., `$array[0]`. `@array` means the whole array “in list context” (Perl-speak, and no I’m not going to try to explain) or length “in scalar context”.
- Much built-in support for working with arrays as lists: “slices”; `push`, `pop`, `shift`, `unshift`; `sort`, `reverse`.

Slide 8

Perl — Hashes

- Expandable collections of key-value pairs, also containing any kind of data.
- Declare with, e.g., `my %hash`, and initialize with list (e.g., `('a', 1, 'b', 2)`) or more explicitly(?) using => (e.g., `('a' => 1, 'b' => 2)`).
- Reference individual elements as, e.g., `$hash{'somekey'}`.
- Get lists of keys or values with `keys`, `values`.

Perl — References

Slide 9

- Motivation for references: No way before they were added to the language to represented nested data structures (e.g., list of lists).
- References are scalars, but their value is a reference to something else, typically an array or hash.
- (Not something I remember details of, but `perldoc perlreftut` is a good introduction.)

Perl — Special Variables

Slide 10

- "Default variable" `$_`. Makes for compact if cryptic code.
- Command-line arguments `@ARGV`.
- Environment variables `%ENV`.
- (Many more, often cryptic.)

Perl — Function Calls

Slide 11

- (Properly speaking, not functions but subroutines.)
- Call function/subroutine with name and arguments in parentheses as in other language, or just with name following by arguments.
- Example: `print` followed by list of things to print.

Perl — Conditional Execution

Slide 12

- Basic syntax familiar but with a twist: `if`, `elif`, `else`. Also `unless`.
- Can also put `if` or `unless` after statement to do conditionally.

Slide 13

Perl — Repetition

- `while` syntax familiar; also `until`.
- C-like `for` but not used much.
- `foreach` on list/array, with or without explicit variable:

  ```
  foreach (@a) { print $_; }
  foreach my $el (@a) { print $el; }
  ```

  Can usefully be combined with `keys` for hashes.

Slide 14

Perl — Operators

- Arithmetic operators familiar from other languages. Note that / is *not* integer division as in many other languages.
- Relational operators — two sets, one for numeric comparisons and one for strings.
- Boolean operators for and, or, not, two versions (C syntax and text names — same meaning but precedence is different). Short-circuit behavior leads to an idiomatic if startling syntax (next slide).
- Familiar syntax for assignments.
- `.` for string concatenation.
- Operator followed by = as in C.

Slide 15

Perl — Files and I/O

- “File handles” can be declared as scalars, or another convention is all-caps global variables. `STDIN`, `STDOUT`, `STDERR` predefined.
- Create with `open`, e.g. (using short-circuiting behavior of `or`):
  `open (INFILE, "<", "in.txt") or die "error";`
  (or replace `INFILE` with `my $in`)
  (`">"` or `">>"` to open for write and append.)
- Read from input file with, e.g., `my $line = <INFILE>;` (Can use this in test of `while`.) (`chomp` to discard end-of-line.)
- Write to output file with `print` with file handle as first parameter.
- `close` as in other languages.

Slide 16

Perl — Pattern Matching

- Good support for pattern matching and substitution. Based on regular expressions, as discussed earlier this semester; same concepts, but as noted syntax details can vary.
- Simple pattern matching for tests:
  `/foo/` true if `$_` contains “foo”.
  `$a ~= /foo/` true if `$a` contains “foo”.
- Simple search-and-replace:
  `s/foo/bar/` or `s/foo/bar/g` to operate on default variable.
  `$a =~ s/foo/bar/` to operate on `$a`.
- As previously, use parentheses to define “capture groups”; reference as `$1`, `$2`, etc.

Slide 17

Perl — Subroutines

- Define with `sub`, e.g., `sub foo { .... }`.
- Call with name followed by comma-separated list of arguments, with or without parentheses.
- By default return nothing; use `return` to return a value (and usual syntax to use it).
- How to declare and use arguments? no way to specify how many or what type, but in subroutine `@_` is list of arguments, so can write, e.g.,
  `my ($a, $b) = @_;`

Slide 18

Perl — Modules

- Perl does provide support for object-oriented programming, via “modules”.
- Defining modules beyond the scope of this lecture.
- Using modules . . .
- Module names generally hierarchical, with components separated by `::`, e.g., `MIME::QuotedPrint`.
- `use` to give access to a module. Most modules have a `man` page with examples of use.
- (An example — many modules in “Library for WWW in Perl” — `perldoc lwptut` for introduction.)

Slide 19

Examples

- (Simple examples.)

Slide 20

Perl as a Scripting Language

- Perl can be useful as “glue” to assemble other programs. Fewer uses than in shell scripts because Perl has so much (more) built in.
- But also supports running external programs — `system` (but does not capture output), “backtick operator” (captures output), `open` with `|` (can pass input, capture output).

Perl “One-Liners”

Slide 21

- Perl, like `sed` and `awk`, can be run with a flag (`-e`) that says “here is the program in the command line”.
- Several examples in documentation (`perldoc perlrun`), concise but fairly inscrutable. Or a Web search for “Perl one-liners” will likely find many examples.

Minute Essay

Slide 22

- Have you worked at all with Perl? How about Python? Do you like these languages in which variables don’t have types? Why or why not?