METAGENOMICS

(and Perl 6)

Excerpted Pages from the eBook available at

https://kyclark.gitbooks.io/metagenomics/content/

Authors and Contributors

- Ken Youens-Clark kyclark@gmail.com
- Bonnie Hurwitz bhurwitz@email.arizona.edu

Table of Contents

Why Perl 6?

Larry Wall (https://en.wikipedia.org/wiki/Larry_Wall) created the Perl programming language around 1987. With what you've seen up to this point, you may be able understand that he essentially cobbled together bash, sed, awk, and grep. He was trying to make a basic Unix systems administration language, and so, like most other Unix languages, it was quite adept at dealing with text. In the mid-1990s, the web was taking off and people were writing CGI (common gateway interface) scripts. Perl was mature enough to be a really good fit for all the text manipulation involved. This was also about the time that bioinformatics was getting started, and most of that is also about dealing with massive quantities of text (e.g., "ACTG").

In 2000, Larry et al. decided to completely revise the language. Much can be said of Perl 5 -- and has -- that it looks like "line noise," that it's incomprehensible even to those who wrote the code, that it's unmaintainable. All this is true! Perl 6 was a completely non-backwards-compatible rewrite of the language. The designers jettisoned all that they thought bad and kept everything they thought good. Whether or not you agree with their decisions, Perl 6 looks to me to be a fine teaching language. Given what you will learn from Perl 6, you will understand "imperative," "declarative," and "functional" programming concepts that you can apply to any future language you teach yourself.

One of the downsides to choosing Perl 6 is that the language is relatively new. Although it has been in development for over 15 years, it was only "officially" release at Christmas 2015 (the long-running joke being that they kept saying it would be done "by Christmas," but they never said which Christmas). So, there are no books and very few examples to draw from on the web. On the other hand, there is a wonderful IRC community on "irc.freenode.net" in the "#perl6" chatroom, and https://docs.perl6.org is pretty comprehensive. Besides, this will build character. Anyway, I'm not trying to make you an expert Perl 6 programmer, just introducing programming concepts with a language that is actually pretty fun to play with.

In the following examples, I will evolve the programs from simpler to more complex to show you how learning higher-order programming techniques actually shorten your code and make it more maintainable. Don't be worried if you don't quite understand the later versions. Just stick with the concepts in the earlier versions and keep reading and playing with the denser functions until you understand them.

With my rationalization complete, let's write some Perl (6)!

Hello, World

You've probably figured out already that the first thing you're supposed to write in any new language is "Hello, World." So, let's do that.

```
$ cat hello.pl6
#!/usr/bin/env perl6
put "Hello, World!"
$ ./hello.pl6
Hello, World!
```

Well, that looks almost exactly like bash and Python except that it uses `put` (like `puts` in Ruby) instead of `echo` or `print`. Maybe this won't be too hard?

OK, let's rewrite our "greeting" script in Perl6.

```
$ cat -n greet1.pl6
     1	#!/usr/bin/env perl6
     2
     3	unless (1 <= @*ARGS.elems <= 2) {
     4	    note "Usage:\n\t{$*PROGRAM-NAME.IO.basename} GREE
TING [NAME]";
     5	    exit 1;
     6	}
     7
     8	my ($greeting, $name) = @*ARGS;
     9
    10	put "$greeting, {$name // 'Stranger'}";
$ ./greet1.pl6 Howdy "Old Joe Clark"
Howdy, Old Joe Clark
```

Well, that looks suspiciously close to the bash version. Line 1 is our now-familiar shebang. We don't need to know the exact path to the Perl6 binary as long as it's somewhere in our $PATH. On line 3, we check the number of arguments to the script by looking at `@*ARGS` which is equivalent to bash's `$@`. In bash we had to write two separate conditionals to check if the number of arguments was with a range, but in Perl we can write it just like in algebra class, "X <= Y <= Z" means Y is greater-than-or-equal to X and less-than-or-equal to Z. The call to `note` is a way to print to STDERR, and we're formatting a standard "Usage" statement. On line 5 we see that we can "exit 1" just as in bash.

One very bash-like thing we see on lines 4 and 10 Line 10 is that we can call a function *inside a string*. In bash, the function call is denoted with `$()` or backticks (``). In Perl curly braces `{}` create what is called a "block" of code that can be executed or used as an argument to another function. Inside the block on line 4, we're using the global/magic variable/object `$*PROGRAM-NAME` to find the `basename` of the program, and on line 10, we're using the `//` operator to say "what's in the $name variable or, if that is not defined, then the string 'Stranger'"). You can read about Perl 6's operators at https://docs.perl6.org/routine.html.

At line 8 is there's a `my`, which is new. Remember in bash how we had to `set -u` to tell bash to complain when we try to use a variable that was never initialized? That kind of checking is built into Perl so that if we were to write `put $greting` it would complain:

```
===SORRY!=== Error while compiling /Users/kyclark/work/abe487/bo
ok/perl6/./greet1.pl6
Variable '$greting' is not declared. Did you mean '$greeting'?
at /Users/kyclark/work/abe487/book/perl6/./greet1.pl6:10
```

Another difference at line 8 is that we can assign two variables, "greeting" and "name," in one line by using a list (the parentheses) on the left-hand side ("LHS" in CS parlance).

Now to talk about the differences in those sigils. We've seen them in bash, and they're in awk and sed, too, but they are usually just `$` signs. Now're we're seeing `$*SPEC` and `@*ARGS`. We need to talk about data shapes.

Data Shapes

Perl has various containers for different shapes of data. You can have one thing (a string of DNA), lots of things (a list of transcription factors), a key-value pair (metadata about a sample like species = 'H. sapiens,' age = 33, gender = 'male'), etc. Perl would call these things "scalars," "lists," and "hashes," and these shapes are almost universal to all programming languages. Perl has several other useful shapes like bags and sets, and we'll get to those later.

Perl's language designers feel that the variables should stand out from the language itself, and that the sigils (decorations on the front, cf. https://en.wikipedia.org/wiki/Sigil_(computer_programming)) on the variables should give the reader an indication of the shape of the data. Languages like Python, Ruby, Haskell, Java, make no visual distinction between their reserved words and variables.

Time to fire up `perl6` and start typing. The following show the result of typing the examples into the REPL.

Scalar $

If you have just one of a thing like our greeting or name, then you put it into a "scalar" or singular variable. These are prefixed with a `$` like `$greeting` and can hold only one value. If you set it to a second value, the first value is forever lost unless you set it to be immutable using `:=` .

```
> my $greeting = "How you doin'?";
How you doin'?
> $greeting.chars
14
> my $money := 1e+6;
1000000
> $money = 0
Cannot assign to an immutable value
  in block <unit> at <unknown file> line 1
```

Array @

When you have an undetermined number of somethings, they belong in an Array (mutable) or a List (immutable). These are plurals, and they start with the `@` sign. The items in a series are separated with commas or you can use the `<>` operator (https://docs.perl6.org/language/quoting). Perl also supports infinite lists -- just don't try to print them.

```
> my @nums = 8, 1, 43;
[8 1 43]
> put @nums.join(', ');
8, 1, 43
> my @sizes = <small medium large>
[small medium large]
> my @x = 1..10
[1 2 3 4 5 6 7 8 9 10]
> my @positive = 1..*
[...]
> @positive[^10]
(1 2 3 4 5 6 7 8 9 10)
```

Hash %

When you have a key-value association, that belongs in a hash AKA "map" or "dictionary" or "associative array," if you're not into the whole brevity thing. Yes, I know we call the `#` a "hash," but this "hash" is short for a "hash table" (https://en.wikipedia.org/wiki/Hash_table). Hashes start with the `%` sign.

```
> my %genome = species => "H_sapiens", taxid => 9606;
{species => H_sapiens, taxid => 9606}
> put %genome<species>;
H_sapiens
> %genome.keys
(species taxid)
```

In the case of each variable, something that should be very striking is that we can ask the variables to do things for us. We can ask a scalar how many "chars" (characters) it has, we can have a list join its elements together using a comma to create a string that we can print, and we can ask the hash to give us the value for some given key or even all the keys it has. We can even go meta (literally) and ask the variables what they can do for us. I will elide the output here, but you should try this in your own REPL:

```
> $greeting.^methods
(BUILD Int Num chomp pred succ simplematch match ...)
> @nums.^methods
(iterator from-iterator new STORE reification-target shape pop s
hift splice ...)
> %genome.^methods
(clone BIND-KEY name keyof of default dynamic push append ...)
```

This only begins to scratch the surface. You can read more at https://docs.perl6.org/type.html.

Scalars

Let's look at some of the things that fit in a scalar. This list isn't exhaustive, just an overture to what we'll see as we move through the text.

Numbers

Perl has many numeric types including Int (integer), UInt (unsigned integer), Num (floating point), Rat/FatRat/Rational, Real, and Complex.

Strings

A "string" (https://docs.perl6.org/type/Str) is a series of characters like "GATTAGA." If I put "2112" in quotes it's a string, but if I don't it's a number. Here's how to create the reverse complement of a strand of DNA:

```
my $dna = "GATTAGA";
> $dna.trans(<A C G T> => <T G C A>).flip
TCTAATC
```

Cool

"Cool" (https://docs.perl6.org/type/Cool) is short for "Convenient OO Loop" and can hold values that look either like numbers or strings, converting back and forth ("coercing," in the parlance) depending on how you use them.

Let's look at a situation where I have a string that I bounce back and forth from being a string or a number:

```
> my $x = "42"
42
> put "$x + 1 = " ~ $x + 1
42 + 1 = 43
```

Lots going on here. First `$x` is a string "42," and in `put "$x + 1 = "` it's treated like a string because it's being interpolated inside quotes. In `$x + 1`, `$x` is coerced to a number because I'm adding it (`+`):

```
> "42" + 1
43
> say ("42" + 1).WHAT
(Int)
```

but then to concantenate it (via `~`) to the other string, the sum of `42 + 1` is coerced to a string:

```
> say ("$x + 1 = " ~ "42" + 1).WHAT
(Str)
```

You can declare that your variables can only contain a certain Type (https://docs.perl6.org/type.html):

```
> my Int $x = "42"
Type check failed in assignment to $x; expected Int but got Str
("42")
  in block <unit> at <unknown file> line 1
> my Str $x = 42;
Type check failed in assignment to $x; expected Str but got Int
(42)
  in block <unit> at <unknown file> line 1
```

But Perl will still coerce values according to how you use them:

```
> my Str $x = "42";
42
> say $x.WHAT
(Str)
> say $x.Int.WHAT
(Int)
> say (+$x).WHAT
(Int)
> put "$x + 1 = " ~ $x + 1
42 + 1 = 43
```

Date, DateTime

The `Date` (https://docs.perl6.org/type/Date) and `DateTime` (https://docs.perl6.org/type/DateTime) types are pretty much what you'd expect:

```
> my $birthday = Date.new(1972, 5, 4);
1972-05-04
> $birthday.day-of-week
4 # Thursday
> DateTime.new(now)
2016-10-31T16:25:09.335788Z
> Date.new(now).year - $birthday.year
44
> my $days = Date.new(now) - $birthday
16251
> $days div 365, $days % 365
(44 191) # (years, days)
```

File handles (IO)

If "foo.txt" is a file that lives on your system, then you can `open` the file to get a file handle that will allow you to read the file. This is a function of I/O (input/output), and lots of interesting types inhabit IO (https://docs.perl6.org/type/IO):

```
my $fh = open "foo.txt";
for $fh.lines -> $line { say $line }
# or
for $fh.lines { .say }
```

Bool

Boolean types (https://docs.perl6.org/type/Bool, named for mathematician George Boole) are your typical `True` and `False` values, but be warned that they are a type of Enum (https://docs.perl6.org/language/typesystem#index-entry-Enumeration-_Enums-_enum), not a truely separate type.

```
> if True { put "true" } else { put "false" }
true
> True == 1
True
> 1 + True == 2
True
```

Pair

A Pair (https://docs.perl6.org/type/Pair) is a combination of a key and a value.

```
> my $bp = A => 'C';
A => C
> $bp.key
A
> $bp.value
C
> $bp.kv
(A C)
```

Proc

A "Proc" (process, https://docs.perl6.org/type/Proc) is the result of running a command outside of Perl such as "pwd":

```
> my $proc = run('pwd', :out)
Proc.new(in => IO::Pipe, out => IO::Pipe.new(:path(""),:chomp),
err => IO::Pipe, exitcode => 0, pid => Any, signal => 0, command
 => ["pwd"])
>
> $proc.out.get
/Users/kyclark
```

Sub

We'll talk about creating subroutines much later, but know that you can store them into a scalar, pass them around like data, and call them when you like.

```
> my $pony = sub { put "I dig a pony" }
sub () { #`(Sub|140377779867312) ... }
> $pony()
I dig a pony
```

Arrays

Arrays are ordered collections of things. Order is important, because later we'll talk about hashes and bags that are unordered. Arrays have lots of handy functions. Let's explore.

Length, Order

How long is that array? Here's a list of the main dogs in my life from when I was a kid to now:

```
> my @dogs = <Chaps Patton Bowzer Logan Lulu Patch>
[Chaps Patton Bowzer Logan Lulu Patch]
> @dogs.elems
6
> +@dogs
6
> @dogs.Int
6
> @dogs.Numeric
6
```

The `elems` methods returns the number of elements and is the most obvious way to find it. Putting a `+` plus sign in front coerces the array into a numerical context which returns the length of the array as does calling the `Int` and `Numeric` methods.

If I want my dogs going from current to first, I can `reverse` them:

```
> @dogs.reverse
[Patch Lulu Logan Bowzer Patton Chaps]
```

I can sort them either by their name:

```
> @dogs.sort
(Bowzer Chaps Logan Lulu Patch Patton)
```

Or, by passing a "unary" operator (one that takes a single argument), by the results of that operation such as the number of characters in the name (i.e., the length of the string):

```
> @dogs.sort(*.chars)
(Lulu Chaps Logan Patch Patton Bowzer)
> @dogs.sort(*.chars).reverse
(Bowzer Patton Patch Logan Chaps Lulu)
```

I can find all possible pairs of dogs:

```
> @dogs.combinations(2)
((Chaps Patton) (Chaps Bowzer) (Chaps Logan) (Chaps Lulu) (Chaps
 Patch) (Patton Bowzer) (Patton Logan) (Patton Lulu) (Patton Pat
ch) (Bowzer Logan) (Bowzer Lulu) (Bowzer Patch) (Logan Lulu) (Lo
gan Patch) (Lulu Patch))
```

I can find just the length of each dog's name by using `map` to apply a the `chars` function to each element:

```
> @dogs.map(*.chars)
(5 6 6 5 4 5)
> @dogs.map(&chars)
(5 6 6 5 4 5)
```

The first version is using the `chars` *method* called on each list object while the second version is applying the `chars` *function* to each element (https://docs.perl6.org/routine/chars). The leading ampersand `&` is passing the `chars` function as a reference. Note that you cannot call it like so:

```
> @dogs.map(chars)
===SORRY!=== Error while compiling:
Calling chars() will never work with proto signature ($)
------> @dogs.map(⏏chars)
```

The `map` function is something I'd really like you to understand, so let's break it down a bit. I can get the same answer (sort of) with a `for` loop:

```
> for @dogs -> $dog { say $dog.chars }
5
6
6
5
4
5
```

And I can capture the numbers by using `do`:

```
> my @chars = do for @dogs -> $dog { $dog.chars }
[5 6 6 5 4 5]
```

Or, more briefly:

```
> my @chars = do for @dogs { .chars }
[5 6 6 5 4 5]
```

So the `map` *function* just turns that around a bit:

```
> my @chars = map { .chars }, @dogs
[5 6 6 5 4 5]
```

And the `map` *method* (of an Array) turns that around again to look more like the `for` version:

```
> my @chars = @dogs.map({ .chars })
[5 6 6 5 4 5]
> my @chars = @dogs.map(*.chars)
[5 6 6 5 4 5]
> my @chars = @dogs.map: *.chars
[5 6 6 5 4 5]
```

You see there is more than one way to write a map. We'll break this down later.

The elements in a Array are not limited to scalars. Using `map`, I create a Array of lists that each combines a dog's name with its length using the `Z` "zip" operator (https://docs.perl6.org/routine/Z):

```
> @dogs Z @dogs.map(*.chars)
((Chaps 5) (Patton 6) (Bowzer 6) (Logan 5) (Lulu 4) (Patch 5))
```

Does that makes sense? Zip takes two lists and combines them element-by-element, stopping on the shorter list:

```
> 1..10 Z 'a'..'z'
((1 a) (2 b) (3 c) (4 d) (5 e) (6 f) (7 g) (8 h) (9 i) (10 j))
> 1..* Z 'Bowzer'.comb
((1 B) (2 o) (3 w) (4 z) (5 e) (6 r))
```

Lastly, I'll show you how `sum` (https://docs.perl6.org/routine/sum) total number of characters in all the dog names:

```
> @dogs.map(*.chars).sum
31
```

Iterating

One of the most common array operations is to iterate over the members while keeping track of the position. Here's a script that breaks a string (here maybe some DNA) into a list using the `comb` method and prints the position and the letter:

```
$ cat -n iterate1.pl6
     1	#!/usr/bin/env perl6
     2
     3	sub MAIN (Str $dna) {
     4	    my $i = 0;
     5	    for $dna.comb -> $letter {
     6	        $i++;
     7	        say "$i: $letter";
     8	    }
     9	}
 $ ./iterate1.pl6 AACTAG
1: A
2: A
3: C
4: T
5: A
6: G
```

This is so common that Arrays have shorter ways to do this:

```
$ cat -n iterate2.pl6
     1	#!/usr/bin/env perl6
     2
     3	sub MAIN (Str $dna) {
     4	    for $dna.comb.kv -> $k, $v {
     5	        say "{$k+1}: $v";
     6	    }
     7	}
$ ./iterate2.pl6 AACTAG
1: A
2: A
3: C
4: T
5: A
6: G
```

Positions in Perl Arrays and Strings start at 0, so I have to add 1 to `$k`. Notice that I can run code *inside a string* by putting `{}` curly braces around it.

I don't have to give the pointy block signature `-> $k, $v` bit. I can use `$^k` and `$^v` (or `$^a` and `$^b` or whatever) to refer to the first and second arguments (in sorted Unicode order) to the block (https://docs.perl6.org/language/variables#index-entry-%24%5E):

```
$ cat -n iterate3.pl6
     1	#!/usr/bin/env perl6
     2
     3	sub MAIN (Str $dna) {
     4	    for $dna.comb.kv { say join ": ", $^k + 1, $^v }
     5	}
$ ./iterate3.pl6 AACTAG
1: A
2: A
3: C
4: T
5: A
6: G
```

Here's a version using `pairs` to get a List of Pair types (https://docs.perl6.org/type/Pair) with the index (position) as the "key" and the letter as the "value":

```
$ cat -n iterate4.pl6
     1	#!/usr/bin/env perl6
     2
     3	sub MAIN (Str $dna) {
     4	    for $dna.comb.pairs -> $pair {
     5	        printf "%s: %s\n", $pair.key + 1, $pair.value;
     6	    }
     7	}
 $ ./iterate3.pl6 AACTAG
1: A
2: A
3: C
4: T
5: A
6: G
```

Again, I don't have to have the `-> $pair` bit if I use the `^` twigil. I can just refer to the one positional argument as `$^pair` and call the `key` and `value` methods on that:

```
$ cat -n iterate5.pl6
     1	#!/usr/bin/env perl6
     2
     3	sub MAIN (Str $dna) {
     4	    for $dna.comb.pairs { say join ': ', $^pair.key +
1, $^pair.value }
     5	}
[saguaro@~/work/metagenomics-book/perl6/lists]$ ./iterate5.pl6 A
ACTAG
1: A
2: A
3: C
4: T
5: A
6: G
```

Or I can use the `:` twigil (https://docs.perl6.org/language/variables#The_:_Twigil) a way to declare named parameters:

```
$ cat -n iterate6.pl6
     1	#!/usr/bin/env perl6
     2
     3	sub MAIN (Str $dna) {
     4	    for $dna.comb.pairs -> (:$key, :$value) {
     5	        say join ': ', $key + 1, $value;
     6	    }
     7	}
$ ./iterate6.pl6 AACTAG
1: A
2: A
3: C
4: T
5: A
6: G
```

Filtering

Often you want to choose or remove certain members of an array. Let's find only the Gs and Cs in a string (https://en.wikipedia.org/wiki/GC-content). Note that I uppercase (`uc`) the `$dna` first so that I only have to check for one case of letters:

```
$ cat -n gc1.pl6
     1	#!/usr/bin/env perl6
     2
     3	sub MAIN (Str $dna) {
     4	    my @gc;
     5	    for $dna.uc.comb -> $base {
     6	        @gc.push($base) if $base eq 'G' || $base eq 'C
';
     7	    }
     8	    say "$dna has {@gc.elems}";
     9	}
$ ./gc1.pl6 AACTAG
AACTAG has 2
```

But `grep` is a much shorter way to find all the elements matching a given condition. Like `map`, `grep` takes a block of code that will be executed for each member of the array. Any elements for which the block evaluates to "True-ish" are allowed through. The `$_` (topic, thing, "it") variable has the current element, so the code is asking "if the thing is a 'G' or if the thing is a 'C'". One can use the `*` to represent "it" and eschew the curly brackets:

```
> grep {$_ > 5}, 1..10
(6 7 8 9 10)
> grep * > 5, 1..10
(6 7 8 9 10)
```

Here's the GC filter written with `grep`:

```
$ cat -n gc2.pl6
     1	#!/usr/bin/env perl6
     2
     3	sub MAIN (Str $dna) {
     4	    my @gc = $dna.uc.comb.grep({$_ eq 'G' || $_ eq 'C'
});
     5	    say "$dna has {@gc.elems}";
     6	}
```

Here I'll use a Junction (https://docs.perl6.org/type/Junction) to compare to "G or C" in one go:

```
$ cat -n gc3.pl6
     1	#!/usr/bin/env perl6
     2
     3	sub MAIN (Str $dna) {
     4	    my @gc = $dna.uc.comb.grep(* eq 'G' | 'C');
     5	    say "$dna has {@gc.elems}";
     6	}
```

Another way to write the `|` Junction is with `any`. The `so` routine (https://docs.perl6.org/routine/so) collapses the various Booleans down to a single value.

```
> 'G' eq 'G' | 'C'
any(True, False)
> so 'G' eq 'G' | 'C'
True
> 'G' eq any(<G C>)
any(True, False)
> so 'G' eq any(<G C>)
True
```

It's extremely common to use regular expressions (https://docs.perl6.org/type/Regex) to filter lists. We'll cover these more later, but here I'm using a character class to represent either "G or C":

```
$ cat -n gc4.pl6
     1	#!/usr/bin/env perl6
     2
     3	sub MAIN (Str $dna) {
     4	    my @gc = $dna.uc.comb.grep(/<[GC]>/);
     5	    say "$dna has {@gc.elems}";
     6	}
```

Here's how you can find the prime numbers between 1 and 10:

```
> (1..10).grep(*.is-prime)
(2 3 5 7)
```

Classification

We can group elements based on predicates we supply. Here is how we can split up the numbers 1 through 10 based on whether they are or are not even divisible by 2:

```
> 2 %% 2
True
> 3 %% 2
False
> (1..10).classify(* %% 2)
{False => [1 3 5 7 9], True => [2 4 6 8 10]}
```

Going back to our G-C counter, we can group each base into whether it is or isn't a "G" or a "C":

```
$ cat -n gc5.pl6
     1	#!/usr/bin/env perl6
     2
     3	sub MAIN (Str $dna) {
     4	    my %hash = $dna.uc.comb.categorize({?/<[GC]>/});
     5	    say "$dna has {%hash<True>.elems}";
     6	}
```

In my opinion, it's not intuitive to use "True" or "False," so let's provide our own String value for the name of the bucket we want:

```
$ cat -n gc6.pl6
     1	#!/usr/bin/env perl6
     2
     3	sub MAIN (Str $dna) {
     4	    my %hash = $dna.uc.comb.categorize({ /<[GC]>/ ?? '
GC' !! 'Other' });
     5	    say "$dna has {%hash<GC>.elems}";
     6	}
```

It might help to see that one in the REPL:

```
> 'AACTAG'.uc.comb.classify({?/<[GC]>/})
{False => [A A T A], True => [C G]}
> 'AACTAG'.uc.comb.classify({/<[GC]>/ ?? 'GC' !! 'Other'})
{GC => [C G], Other => [A A T A]}
```

`classify` takes a code block and uses the resulting string to put the element into a bucket. Here I've used the same regular expression `/<[GC]>/` to return the string "GC" if it's a match or "Other" if it's not. The combination of the `?? !!` is the "ternary" operator that we'll talk about more later. The resulting Hash has a key called "GC" and its value is a list containing the "G" and "C" found in the string.

So you're seeing that lists can be inside of other Lists as well as inside of Hashes and other data structures.

I can classify my `@dogs` based on the length of their names using that same syntax variations we saw for `map` :

```
> @dogs.classify({.chars})
{4 => [Lulu], 5 => [Chaps Logan Patch], 6 => [Patton Bowzer]}
> @dogs.classify(*.chars)
{4 => [Lulu], 5 => [Chaps Logan Patch], 6 => [Patton Bowzer]}
> @dogs.classify(&chars)
{4 => [Lulu], 5 => [Chaps Logan Patch], 6 => [Patton Bowzer]}
```

Lists can also be composed of Pairs (https://docs.perl6.org/type/Pair). Here I'll redeclare my `@dogs` with their names as the "key" and thier sex as the "value." Then I can `classify` them on their `value` :

```
> my @dogs = Chaps => 'male', Patton => 'male', Bowzer => 'male'
, Logan => 'male', Lulu => 'female', Patch => 'male'
[Chaps => male Patton => male Bowzer => male Logan => male Lulu
=> female Patch => male]
> @dogs.classify(-> $dog {$dog.value})
{female => [Lulu => female], male => [Chaps => male Patton => ma
le Bowzer => male Logan => male Patch => male]}
> @dogs.classify({$^dog.value})
{female => [Lulu => female], male => [Chaps => male Patton => ma
le Bowzer => male Logan => male Patch => male]}
> @dogs.classify({.value})
{female => [Lulu => female], male => [Chaps => male Patton => ma
le Bowzer => male Logan => male Patch => male]}
> @dogs.classify(*.value)
{female => [Lulu => female], male => [Chaps => male Patton => ma
le Bowzer => male Logan => male Patch => male]}
```

Picking random elements

To finish off, let's write a Shakespearean insult generator that uses the `pick` method to randomly choose some perjoratives:

```
$ cat -n insult.pl6
     1	#!/usr/bin/env perl6
     2
     3	sub MAIN (Int :$n=1) {
     4	    my @adjectives = qw{scurvy old filthy scurilous la
scivious
     5	        foolish rascaly gross rotten corrupt foul loat
hsome irksome
     6	        heedless unmannered whoreson cullionly false f
ilthsome
     7	        toad-spotted caterwauling wall-eyed insatiate
vile peevish
     8	        infected sodden-witted lecherous ruinous indis
tinguishable
     9	        dishonest thin-faced slanderous bankrupt base
detestable
    10	        rotten dishonest lubbery};
    11	    my @nouns = qw{knave coward liar swine villain beg
gar
    12	        slave scold jolthead whore barbermonger fishmo
nger carbuncle
    13	        fiend traitor block ape braggart jack milksop
boy harpy
    14	        recreant degenerate Judas butt cur Satan ass c
oxcomb dandy
    15	        gull minion ratcatcher maw fool rogue lunatic
varlet worm};
    16
    17	    printf "You %s, %s, %s %s!\n",
    18	        @adjectives.pick(3), @nouns.pick for ^$n;
    19	}
$ ./insult.pl6 -n=5
You foul, dishonest, old recreant!
You irksome, gross, false degenerate!
You old, dishonest, toad-spotted jack!
You ruinous, unmannered, foolish slave!
You scurry, slanderous, peevish harpy!
```

You can also use `roll` so that each selection is made independently. Above I'm using the `qw{}` "quote-word" operator (https://docs.perl6.org/language/quoting#index-entry-qw_word_quote) to create a list of words rather than writing:

```
my @adjectives = "scurvy", "old", "filthy";
```

You should spend a few minutes reading about all the different quoting options available as they will come in handy.

Hashes

Hashes (https://docs.perl6.org/type/Hash) basically Arrays of Pairs (https://docs.perl6.org/type/Pair). They are also known as dictionaries or maps. The key in a Hash is usually a string, and the value can be pretty much anything. The order of the Pairs is not preserved in creation order:

```
> my %dog = name => 'Patch', age => 4, color => 'white', weight
=> '31 lbs'
{age => 4, color => white, name => Patch, weight => 31 lbs}
```

If I ask for one key, I get one value:

```
> %dog<name>
Patch
```

I can also ask for more than one key, and I will get a list:

```
> %dog<age color>
(4 white)
```

I can ask for all the keys and values using the `kv` method:

```
> %dog.kv
(color white name Patch age 4 weight 31 lbs)
> %dog.pairs
(color => white name => Patch age => 4 weight => 31 lbs)
> say join ", ", %dog.map(*.kv.join(": "))
color: white, name: Patch, age: 4, weight: 31 lbs
> %dog.pairs.map(*.kv.join(": ")).join(", ")
color: white, name: Patch, age: 4, weight: 31 lbs
```

If I want the keys back in a particular order, I either have to ask for them explicitly or `sort` them:

```
> %dog.keys
(color name age weight)
> %dog.keys.sort
(age color name weight)
```

I can store more than one thing for a value by using lists as in this amino-acid-to-codons table:

```
$ cat -n aa.pl6
     1	#!/usr/bin/env perl6
     2
     3	my %aa = Isoleucine    => <ATT ATC ATA>,
     4	         Leucine       => <CTT CTC CTA CTG TTA TTG>,
     5	         Valine        => <GTT GTC GTA GTG>,
     6	         Phenylalanine => <TTT TTC>,
     7	         Methionine    => <ATG>,
     8	         Cysteine      => <TGT TGC>,
     9	         Alanine       => <GCT GCC GCA GCG>,
    10	         Glycine       => <GGT GGC GGA GGG>,
    11	         Proline       => <CCT CCC CCA CCG>,
    12	         Threonine     => <ACT ACC ACA ACG>,
    13	         Serine        => <TCT, TCC, TCA, TCG, AGT, AG
C>,
    14	         Tyrosine      => <TAT TAC>,
    15	         Tryptophan    => <TGG>,
    16	         Glutamine     => <CAA CAG>,
    17	         Asparagine    => <AAT AAC>,
    18	         Histidine     => <CAT CAC>,
    19	         Glutamic_acid => <GAA GAG>,
    20	         Aspartic_acid => <GAT GAC>,
    21	         Lysine        => <AAA AAG>,
    22	         Arginine      => <CGT CGC CGA CGG AGA AGG>,
    23	         Stop          => <TAA TAG TGA>;
    24
    25	for %aa.keys.sort -> $key {
    26	    printf "%-15s = %s\n", $key, %aa{ $key }.join(", "
);
    27	}
$ ./aa.pl6
```

```
Alanine         = GCA, GCC, GCG, GCT
Arginine        = AGA, AGG, CGA, CGC, CGG, CGT
Asparagine      = AAC, AAT
Aspartic_acid   = GAC, GAT
Cysteine        = TGC, TGT
Glutamic_acid   = GAA, GAG
Glutamine       = CAA, CAG
Glycine         = GGA, GGC, GGG, GGT
Histidine       = CAC, CAT
Isoleucine      = ATA, ATC, ATT
Leucine         = CTA, CTC, CTG, CTT, TTA, TTG
Lysine          = AAA, AAG
Methionine      = ATG
Phenylalanine   = TTC, TTT
Proline         = CCA, CCC, CCG, CCT
Serine          = AGC, AGT,, TCA,, TCC,, TCG,, TCT,
Stop            = TAA, TAG, TGA
Threonine       = ACA, ACC, ACG, ACT
Tryptophan      = TGG
Tyrosine        = TAC, TAT
Valine          = GTA, GTC, GTG, GTT
```

Hashes are great for counting things. You don't have to check if a key/value pair exists first -- just add one to a key and it will be created and start counting from 0:

```
$ cat -n word-count.pl6
     1	#!/usr/bin/env perl6
     2
     3	sub MAIN (Str $file where *.IO.f) {
     4	    my %count;
     5	    for $file.IO.words -> $word {
     6	        %count{ $word }++;
     7	    }
     8
     9	    for %count.grep(*.value > 1) -> (:$key, :$value) {
    10	        printf "Saw '%s' %s time.\n", $key, $value;
    11	    }
    12	}
$ ./word-count.pl6 because.txt
Saw 'a' 3 times.
Saw 'of' 2 times.
Saw 'for' 2 times.
Saw 'I' 3 times.
Saw 'The' 3 times.
Saw 'but' 3 times.
Saw 'And' 2 times.
Saw 'We' 5 times.
Saw 'passed' 3 times.
Saw 'the' 6 times.
```

You can set the keys and values all-at-once like above or one-at-a-time:

```
$ cat -n bridge-of-death.pl6
     1    #!/usr/bin/env perl6
     2
     3    my %answers;
     4    for "name", "quest", "favorite color" -> $key {
     5        %answers{ $key } = prompt "What is your $key? ";
     6    }
     7
     8    if %answers{"favorite color"} eq "Blue" {
     9        put "Right.  Off you go.";
    10    }
    11    else {
    12        put "AAAAAAAAAAaaaaaaaaa!"
    13    }
$ ./bridge-of-death.pl6
What is your name? My name is Sir Launcelot of Camelot
What is your quest? To seek the Holy Grail
What is your favorite color? Blue
Right.  Off you go.
$ ./bridge-of-death.pl6
What is your name? Sir Galahad of Camelot.
What is your quest? I seek the Holy Grail.
What is your favorite color? Blue.  No yel--
AAAAAAAAAAaaaaaaaaa!
```

Sometimes you need to see if a key is defined in a hash, and for that you use the adverb `:exists`:

```
$ cat -n gashlycrumb.pl6
     1    #!/usr/bin/env perl6
     2
     3    # Text by Edward Gorey
     4
     5    my %alphabet = q:to/END/.lines.map(-> $line { $line.su
bstr(0,1) => $line });
     6    A is for Amy who fell down the stairs.
     7    B is for Basil assaulted by bears.
     8    C is for Clara who wasted away.
     9    D is for Desmond thrown out of a sleigh.
```

```
    10	E is for Ernest who choked on a peach.
    11	F is for Fanny sucked dry by a leech.
    12	G is for George smothered under a rug.
    13	H is for Hector done in by a thug.
    14	I is for Ida who drowned in a lake.
    15	J is for James who took lye by mistake.
    16	K is for Kate who was struck with an axe.
    17	L is for Leo who choked on some tacks.
    18	M is for Maud who was swept out to sea.
    19	N is for Neville who died of ennui.
    20	O is for Olive run through with an awl.
    21	P is for Prue trampled flat in a brawl.
    22	Q is for Quentin who sank on a mire.
    23	R is for Rhoda consumed by a fire.
    24	S is for Susan who perished of fits.
    25	T is for Titus who flew into bits.
    26	U is for Una who slipped down a drain.
    27	V is for Victor squashed under a train.
    28	W is for Winnie embedded in ice.
    29	X is for Xerxes devoured by mice.
    30	Y is for Yorick whose head was bashed in.
    31	Z is for Zillah who drank too much gin.
    32	END
    33
    34	loop {
    35	    my $letter = uc prompt "Letter [0 to quit]? ";
    36
    37	    if $letter eq '0' {
    38	        put "Bye.";
    39	        exit;
    40	    }
    41	    elsif %alphabet{ $letter }:exists {
    42	        put %alphabet{ $letter };
    43	    }
    44	    else {
    45	        put "I'm sorry, I don't know that letter ($let
ter).";
    46	    }
    47	}
$ ./gashlycrumb.pl6
```

```
Letter [0 to quit]? j
J is for James who took lye by mistake.
Letter [0 to quit]? 9
I'm sorry, I don't know that letter (9).
Letter [0 to quit]? 0
Bye.
```