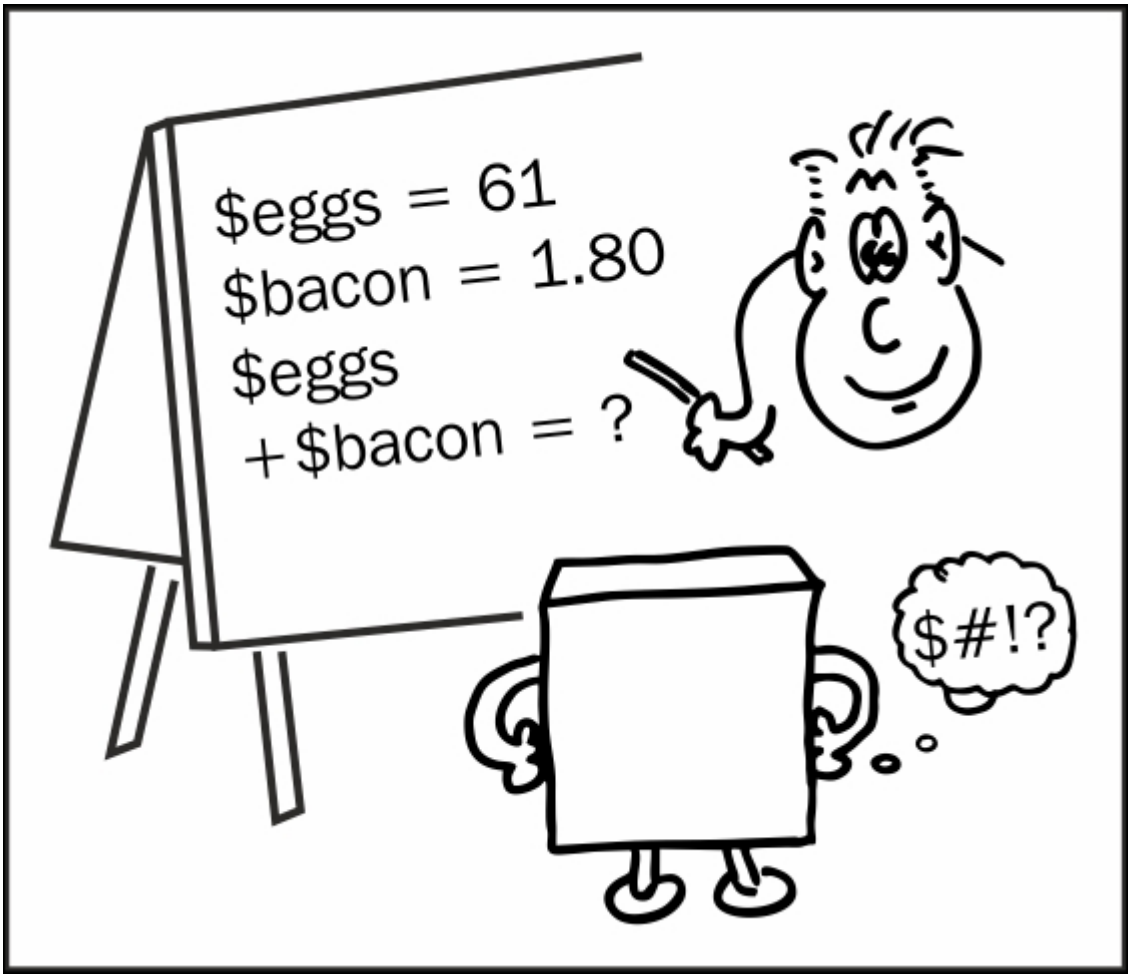
Lab 5. Variables, expressions, and facts

[*It is impossible to begin to learn that which one thinks one already knows.*]

--[[*Epictetus*]]{.attribution}

In this lab, you will learn about Puppet variables and data types, expressions, and conditional statements. You will also learn how Puppet manifests can get data about the node using Facter, find out which are the most important standard facts, and see how to create your own external facts. Finally, you will use Puppet's `each` function to iterate over arrays and hashes, including Facter data.



Introducing variables

A **variable** in Puppet is simply a way of giving a name to a particular value, which we could then use wherever we would use the literal value (`variable_string.pp`):

```
$php_package = 'php7.0-cli'

package { $php_package:
  ensure => installed,
}
```

The dollar sign (`$`) tells Puppet that what follows is a variable name. Variable names must begin with a lowercase letter or an underscore, though the rest of the name can also contain uppercase letters or numbers.

A variable can contain different types of data; one such type is a **String** (like `php7.0-cli`), but Puppet variables can also contain **Number** or **Boolean** values (`true` or `false`). Here are a few examples (`variable_simple.pp`):

```
$my_name = 'Zaphod Beeblebrox'
$answer = 42
$scheduled_for_demolition = true
```

Using Booleans

Strings and numbers are straightforward, but Puppet also has a special data type to represent true or false values, which we call **Boolean** values, after the logician George Boole. We have already encountered some Boolean values in Puppet resource attributes (`service.pp`):

```
service { 'sshd':
  ensure => running,
  enable => true,
}
```

The only allowed values for Boolean variables are the literal values `true` and `false` , but Boolean variables can also hold the values of conditional expressions (expressions whose value is `true` or `false`), which we'll explore later in this lab.

Note

You might be wondering what type the value `running` is in the previous example. It's actually a string, but a special, unquoted kind of string called a **bare word**. Although []{#id178 .indexterm}it would be exactly the same to Puppet if you used a normal quoted string `'running'` here, it's considered good style to use bare words for attribute values which can only be one of a small number of words (for example, the `ensure` attribute on services can only take the values `running` or `stopped`). By contrast, `true` is not a bare word but a Boolean value, and it is not interchangeable with the string `'true'` . Always use the unquoted literal values `true` or `false` for Boolean values.

Interpolating variables in strings

It's no good being able to store something in a variable if you can't get it out again, and one of the most common ways to use a variable's value is to **interpolate** it in a string. When you do this, Puppet inserts the current value of the variable into the contents of the string, replacing the name of the variable. String interpolation looks like this (`string_interpolation.pp`):

```
$my_name = 'John'
notice("Hello, ${my_name}! It's great to meet you!")
```

When you apply this manifest, the following output is printed:

```
Notice: Scope(Class[main]): Hello, John! It's great to meet you!
```

To interpolate (that is, to insert the value of) a variable in a string, prefix its name with a `$` character and surround it with curly braces (`{}`). This tells Puppet to replace the variable's name with its value in the string.

Note

We sneaked a new Puppet function, `notice()` , into the previous example. It has no effect on the system, but it prints out the value of its argument. This can be very useful for troubleshooting problems or finding out what the value of a variable is at a given point in your manifest.

Creating arrays

A variable can also hold more than one value. An **Array** is an ordered sequence of values, each of which can be of any type. The following example creates an array of **Integer** values (`variable_array.pp`):

```
$heights = [193, 120, 181, 164, 172]

$first_height = $heights[0]
```

You can refer to any individual element of an array by giving its index number in square brackets, where the first element is index `[0]` , the second is `[1]` , and so on. (If you find this confusing, you're not alone, but it may help to think of the index as representing an offset from the beginning of the array. Naturally, then, the offset of the first element is 0.)

Declaring arrays of resources

You already know that in Puppet resource declarations, the title of the resource is usually a string, such as the path to a file or the name of a package. You might as well ask, "What happens if you supply an array of strings as the title of a resource instead of a single string? Does Puppet create multiple resources, one for each element in the array?" Let's try an experiment where we do exactly that with an array of package names and see what happens (`resource_array.pp`):

```
$dependencies = [
  'php7.0-cgi',
  'php7.0-cli',
  'php7.0-common',
  'php7.0-gd',
  'php7.0-json',
  'php7.0-mcrypt',
  'php7.0-mysql',
  'php7.0-soap',
]

package { $dependencies:
  ensure => installed,
}
```

If our intuition is right, applying the previous manifest should give us a package resource for each package listed in the `$dependencies` array, and each one should be installed. Here's what happens when the manifest is applied:

```
sudo apt-get update
sudo puppet apply /examples/resource_array.pp
```

```
Notice: Compiled catalog for ubuntu-xenial in environment production in 0.68 seconds
Notice: /Stage[main]/Main/Package[php7.0-cgi]/ensure: created
Notice: /Stage[main]/Main/Package[php7.0-cli]/ensure: created
Notice: /Stage[main]/Main/Package[php7.0-common]/ensure: created
Notice: /Stage[main]/Main/Package[php7.0-gd]/ensure: created
Notice: /Stage[main]/Main/Package[php7.0-json]/ensure: created
Notice: /Stage[main]/Main/Package[php7.0-mcrypt]/ensure: created
Notice: /Stage[main]/Main/Package[php7.0-mysql]/ensure: created
Notice: /Stage[main]/Main/Package[php7.0-soap]/ensure: created
Notice: Applied catalog in 56.98 seconds
```

Giving an array of strings as the title of a resource results in Puppet creating multiple resources, all identical except for the title. You can do this not just with packages, but also with files, users, or, in fact, any type of resource. We'll see some more sophisticated ways of creating resources from data in [Lab 6], [*Managing data with Hiera*].

Note

Why did we run `sudo apt-get update` before applying the manifest? This is the Ubuntu command to update the system's local package catalog from the upstream servers. It's []{#id186 .indexterm}always a good idea to run this before installing any package to make sure you're installing the latest version. In your production Puppet code, of course, you can run this via an `exec` resource.

Understanding hashes

A **hash**, also known as a dictionary in some programming languages, is like an array, but []{#id188 .indexterm}instead of just being a sequence of values, each value has a name (`variable_hash.pp`):

```
$heights = {
  'john'    => 193,
  'rabiah'  => 120,
  'abigail' => 181,
  'melina'  => 164,
  'sumiko'  => 172,
}

notice("John's height is ${heights['john']}cm.")
```

The name for each value is known as the **key**. In the previous example, the keys of this hash are `john` , `rabiah` , `abigail` , `melina` , and `sumiko` . To look up the value of a given key, you put the key in square brackets after the hash name: `$heights['john']` .

Note

Puppet style note

Did you spot the trailing comma on the last hash key-value pair and the last element of the array in the previous example? Although the comma isn't strictly required, it's good style to add one. The reason is that it's very common to want to add another item to an array or hash, and if your last item already has a trailing comma, you won't have to remember to add one when extending the list.

Setting resource attributes from a hash

You might have noticed that a hash looks a lot like the attributes of a resource: it's a one-to-one mapping between names and values. Wouldn't it be convenient if, when declaring resources, we could just specify a hash containing all

the attributes and their values? As it happens, you can do just that (`hash_attributes.pp`):

```
$attributes = {
  'owner' => 'ubuntu',
  'group' => 'ubuntu',
  'mode'  => '0644',
}

file { '/tmp/test':
  ensure => present,
  *      => $attributes,
}
```

The `*` character, cheerfully named the **attribute splat operator**, tells Puppet to treat the specified hash as a list of attribute-value pairs to []{#id192 .indexterm}apply to the resource. This is exactly equivalent to specifying the same attributes directly, as in the following example:

```
file { '/tmp/test':
  ensure => present,
  owner  => 'vagrant',
  group  => 'vagrant',
  mode   => '0644',
}
```

Introducing expressions

Variables are not the only things in Puppet that have a value. Expressions also have a value. The simplest expressions are just literal values:

```
42
true
'Oh no, not again.'
```

You can combine numeric values with arithmetic operators, such as `+` , `-` , `*` , and `/` , to create **arithmetic**] **expressions**, which have a numeric value, and you can use these to have Puppet do calculations (`expression_numeric.pp`):

```
$value = (17 * 8) + (12 / 4) - 1
notice($value)
```

The most useful expressions, though, are which that evaluate to `true` or `false` , known as **Boolean expressions**. The following is a set of examples of Boolean expressions, all of which evaluate to `true` (`expression_boolean.pp`):

```
notice(9 < 10)
notice(11 > 10)
notice(10 >= 10)
notice(10 <= 10)
notice('foo' == 'foo')
notice('foo' in 'foobar')
notice('foo' in ['foo', 'bar'])
notice('foo' in { 'foo' => 'bar' })
```

```
notice('foo' =~ /oo/)
notice('foo' =~ String)
notice(1 != 2)
```

Meeting Puppet's comparison operators

All the operators in the Boolean expressions shown in the previous example are known as **comparison operators**, because they compare two []{#id197 .indexterm}values. The result is either `true` or `false`. These are the comparison operators Puppet provides:

- `==` and `!=` (equal, not equal)
- `>`, `>=`, `<`, and `<=` (greater than, greater than or equal to, less than, less than or equal to)
- `A in B` (`A` is a substring of `B`, `A` is an element of the array `B`, or `A` is a key of the hash `B`)
- `A =~ B` (`A` is matched by the regular expression `B`, or `A` is a value of data type `B`. For example, the expression `'hello' =~ String` is `true`, because the value `'hello'` is of type String.)

Introducing regular expressions

The `=~` operator tries to match a given value against a **regular expression**. A regular expression ([*regular*] in the sense of constituting a pattern or a rule) is a special kind of expression which specifies a set of strings. For example, the regular expression `/a+/` describes the set of all strings that contain one or more consecutive `a`s: `a`, `aa`, `aaa`, and so on, as well as all strings which contain such a sequence among other characters. The slash characters `//` delimit a regular expression in Puppet.

When we say a regular expression **matches** a value, we mean the value is one of the set of strings specified by the regular expression. The regular expression `/a+/` would match the string `aaa` or the string `Aaaaargh!`, for example.

The following example shows some regular expressions that match the string `foo` (`regex.pp`):

```
$candidate = 'foo'
notice($candidate =~ /foo/) # literal
notice($candidate =~ /f/)   # substring
notice($candidate =~ /f.*/) # f followed by zero or more characters
notice($candidate =~ /f.o/) # f, any character, o
notice($candidate =~ /fo+/) # f followed by one or more 'o's
notice($candidate =~ /[fgh]oo/) # f, g, or h followed by 'oo'
```

Note

Regular expressions are more-or-less a standard language for expressing string patterns. It's a complicated and powerful language, which really deserves a course of its own (and there are several), but suffice it to say for now that Puppet's regular expression syntax is the same as that used in the Ruby language. You can read more about it in the Ruby documentation at:

http://ruby-doc.org/core/Regexp.html

Using conditional expressions

Boolean expressions, like those in the previous example, are useful because we can use them to make choices in the Puppet manifest. We can apply certain resources only if a given condition is met, or we can assign an attribute one

value or another, depending on whether some expression is true. An expression used in this way is called a **conditional expression**.

Making decisions with if statements

The most common use of a conditional expression is in an `if` statement. The following example shows how to use `if` to decide whether to apply a resource (`if.pp`):

```
$install_perl = true
if $install_perl {
  package { 'perl':
    ensure => installed,
  }
} else {
  package { 'perl':
    ensure => absent,
  }
}
```

You can see that the value of the Boolean variable `$install_perl` governs whether or not the `perl` package is installed. If `$install_perl` is `true` , Puppet will apply the following resource:

```
  package { 'perl':
    ensure => installed,
  }
```

If, on the other hand, `$install_perl` is `false` , the resource applied will be:

```
  package { 'perl':
    ensure => absent,
  }
```

You can use `if` statements to control the application of any number of resources or, indeed, any part of your Puppet manifest. You can leave out the `else` clause if you like; in that case, when the value of the conditional expression is `false` , Puppet will do nothing.

Choosing options with case statements

The `if` statement allows you to take a yes/no decision based on the value of a Boolean expression. But if you need to make a choice among more than two options, you can use a `case` statement instead (`case.pp`):

```
$webserver = 'nginx'
case $webserver {
  'nginx': {
    notice("Looks like you're using Nginx! Good choice!")
  }
  'apache': {
    notice("Ah, you're an Apache fan, eh?")
  }
  'IIS': {
    notice('Well, somebody has to.')
  }
  default: {
```

```
    notice("I'm not sure which webserver you're using!")
  }
}
```

In a `case` statement, Puppet compares the value of the expression to each of the cases listed in order. If it finds a match, the corresponding resources are applied. The special case called `default` always matches, and you can use it to make sure that Puppet will do the right thing even if none of the other cases match.

Finding out facts

It's very common for Puppet manifests to need to know something about the system they're running on, for example, its hostname, IP address, or operating system version. Puppet's built-in mechanism for getting system information is called **Facter**, and each piece of information provided by Facter is known as a **fact**.

Using the facts hash

You can access Facter facts in your manifest using the **facts hash**. This is a Puppet []{#id211 .indexterm}variable called `$facts` which is available everywhere in the manifest, and to get a particular fact, you supply the name of the fact you want as the key (`facts_hash.pp`):

```
notice($facts['kernel'])
```

On the Vagrant box, or any Linux system, this will return the value `Linux` .

In older versions of Puppet, each fact was a distinct global variable, like this:

```
notice($::kernel)
```

You will still see this style of fact reference in some Puppet code, though it is now deprecated and will eventually stop working, so you should always use the $facts hash instead.

Running the facter command

You can also use the `facter` command to see the value of particular facts, or just see what facts are available. For example, running `facter os` on the command line will show you the hash of available OS-related facts:

```
facter os
{
  architecture => "amd64",
  distro => {
    codename => "xenial",
    description => "Ubuntu 16.04 LTS",
    id => "Ubuntu",
    release => {
      full => "16.04",
      major => "16.04"
    }
  },
  family => "Debian",
  hardware => "x86_64",
  name => "Ubuntu",
  release => {
    full => "16.04",
```

```
    major => "16.04"
  },
  selinux => {
    enabled => false
  }
}
```

You can also use the `puppet facts` command to see what facts will be available to Puppet manifests. This will [] {#id214 .indexterm}also include any custom facts defined by []{#id215 .indexterm}third-party Puppet modules (see [Lab 7], [*Mastering modules*], for more information about this).

Accessing hashes of facts

As in the previous example, many facts actually return a hash of values, rather than a single value. The value of the `$facts['os']` fact is a hash with the keys `architecture`, `distro`, `family`, `hardware`, `name`, `release`, and `selinux`. Some of those are also hashes; it's hashes all the way down!

As you know, to access a particular value in a hash, you specify the key name in square brackets. To access a value inside a hash, you add another key name in square brackets after the first, as in the following example (`facts_architecture.pp`):

```
notice($facts['os']['architecture'])
```

You can keep on appending more keys to get more and more specific information (`facts_distro_codename.pp`):

```
notice($facts['os']['distro']['codename'])
```

Note

Key fact

The operating system major release is a very handy fact and one you'll probably use often:

```
$facts['os']['release']['major']
```

Referencing facts in expressions

Just as with ordinary variables or values, you can use facts in expressions, including conditional expressions (`fact_if.pp`):

```
if $facts['os']['selinux']['enabled'] {
  notice('SELinux is enabled')
} else {
  notice('SELinux is disabled')
}
```

Note

Although conditional expressions based on facts can be useful, an even better way of making decisions based on facts in your manifests is to use Hiera, which we'll cover in the next lab. For example, if you find yourself writing an `if` or `case` statement which chooses different resources depending on the operating system version, consider using a Hiera query instead.

Using memory facts

Another useful set of facts is that relating to the **system memory**. You can find out the total physical memory available, and the amount of memory currently used, as well as the same figures for swap memory.

One common use for this is to configure applications dynamically based on the amount of system memory. For example, the MySQL parameter `innodb_buffer_pool_size` specifies the amount of memory allocated to database query cache and indexes, and it should generally be set as high as possible ("[*as large a value as practical, leaving enough memory for other processes on the node to run without excessive paging*]", according to the documentation). So you might decide to set this to three-quarters of total memory (for example), using a fact and an arithmetic expression, as in the following snippet (`fact_memory.pp`):

```
$buffer_pool = $facts['memory']['system']['total_bytes'] * 3/4
notice("innodb_buffer_pool_size=${buffer_pool}")
```

Note

Key fact

The total system memory fact will help you calculate configuration parameters which vary as a fraction of memory:

```
$facts['memory']['system']['total_bytes']
```

Discovering networking facts

Most applications use the network, so you'll find Facter's network-related facts very useful for anything to do with network configuration. The most commonly used facts are the system hostname, fully qualified domain name (FQDN), and IP address (`fact_networking.pp`):

```
notice("My hostname is ${facts['hostname']}")
notice("My FQDN is ${facts['fqdn']}")
notice("My IP is ${facts['networking']['ip']}")
```

Note

Key fact

The system hostname is something you'll need to refer to often in your manifests:

```
$facts['hostname']
```

Providing external facts

While the built-in facts available to Puppet provide a lot of important information, you can make the `$facts` hash even more useful by extending it with []{#id226 .indexterm}your own facts, known as **external facts**. For example, if nodes are located in different cloud providers, each of which requires a slightly different networking setup, you could create a custom fact called `cloud` to document this. You can then use this fact in manifests to make decisions.

Puppet looks for external facts in the `/opt/puppetlabs/facter/facts.d/` directory. Try creating a file in that directory called `facts.txt` with the following contents (`fact_external.txt`):

```
cloud=aws
```

A quick way to do this is to run the following command:

```
sudo cp /examples/fact_external.txt /opt/puppetlabs/facter/facts.d
```

The `cloud` fact is now available in your manifests. You can check that the fact is working by running the following command:

```
sudo facter cloud
aws
```

To use the fact in your manifest, query the `$facts` hash just as you would for a built-in fact (`fact_cloud.pp`):

```
case $facts['cloud'] {
  'aws': {
    notice('This is an AWS cloud node ')
  }
  'gcp': {
    notice('This is a Google cloud node')
  }
  default: {
    notice("I'm not sure which cloud I'm in!")
  }
}
```

You can put as many facts in a single text file as you like, or you can have each fact in a separate file: it doesn't make any difference. Puppet will read all the files in the `facts.d/` directory and extract all the `key=value` pairs from each one.

Text files work well for simple facts (those that return a single value). If your external facts need to return structured data (arrays or hashes, for example), you can use a YAML or JSON file instead to do this. We'll be learning more about YAML in the next lab, but for now, if you need to build structured external facts, consult the Puppet documentation for details.

It's common to set up external facts like this at build time, perhaps as part of an automated bootstrap script (see [Lab 12], [*Putting it all together*], for more about the bootstrap process).

Creating executable facts

External facts are not limited to static text files. They can also be the output of scripts or programs. For example, you could write a script that calls a web service to get some data, and the result would be the value of the fact. These are known as **executable facts**.

Executable facts live in the same directory as other external facts (`/opt/puppetlabs/facter/facts.d/)` , but they are distinguished by having the execute bit set on their files (recall that files on Unix-like systems each have a set of bits indicating their read, write, and execute permissions) and they also can't be named with `.txt` , `.yaml` , or `.json` extensions. Let's build an executable fact which simply returns the current date, as an example:

1. Run the following command to copy the executable fact example into the external fact directory:

   ```
   sudo cp /examples/date.sh /opt/puppetlabs/facter/facts.d
   ```

2. Set the execute bit on the file with the following command:

   ```
   sudo chmod a+x /opt/puppetlabs/facter/facts.d/date.sh
   ```

3. Now test the fact:

```
sudo facter date
2017-04-12
```

Here is the script which generates this output (`date.sh`):

```
#!/bin/bash
echo "date=`date +%F`"
```

Note that the script has to output `date=` before the actual date value. This is because Facter expects executable facts to output a list of `key=value` pairs (just one such pair, in this case). The `key` is the name of the fact (`date`), and the `value` is whatever is returned by `` `date +%F` `` (the current date in ISO 8601 format). You should use ISO 8601 format (`YYYY-MM-DD`) whenever you need to represent dates, by the way, because it's not only the international standard date format, but it is also unambiguous and sorts alphabetically.

As you can see, executable facts are quite powerful because they can return any information which can be generated by a program (the program could make network requests or database queries, for example). However, you should use executable facts with care, as Puppet has to evaluate [*all*] external facts on the node every time it runs, which means running every script in `/opt/puppetlabs/facter/facts.d` .

Note

If you don't need the information from an executable fact to be regenerated every time Puppet runs, consider running the script from a cron job at longer intervals and having it write output to a static text file in the facts directory instead.

Iterating over arrays

Iteration (doing something repeatedly) is a useful technique in your Puppet manifests to avoid lots of duplicated code. For example, consider the following manifest, which creates several files with identical properties (`iteration_simple.pp`):

```
file { '/usr/local/bin/task1':
  content => "echo I am task1\n",
  mode    => '0755',
}

file { '/usr/local/bin/task2':
  content => "echo I am task2\n",
  mode    => '0755',
}

file { '/usr/local/bin/task3':
  content => "echo I am task3\n",
  mode    => '0755',
}
```

You can see that each of these resources is identical, except for the task number: `task1` , `task2` , and `task3` . Clearly, this is a lot of typing and should you later decide to change the properties of these scripts (for example, moving them to a different directory), you'll have to find and change each one in the manifest. For three resources, this is already annoying, but for thirty or a hundred resources it's completely impractical. We need a better solution.

Using the each function

Puppet provides the `each` function to help with just this kind of situation. The `each` function takes an array and applies a block of Puppet code to each element of the array. Here's the same example we saw previously, only this time using an array and the `each` function (`iteration_each.pp`):

```
$tasks = ['task1', 'task2', 'task3']
$tasks.each | $task | {
  file { "/usr/local/bin/${task}":
    content => "echo I am ${task}\n",
    mode    => '0755',
  }
}
```

Now this looks more like a computer program! We have a **loop**, created by the `each` function. The loop goes round and round, creating a new `file` resource for each element of the `$tasks` array. Let's look at a schematic version of an `each` loop:

```
ARRAY.each | ELEMENT | {
  BLOCK
}
```

The following list describes the components of the `each` loop:

- `ARRAY` can be any Puppet array variable or literal value (it could even be a call to Hiera that returns an array). In the previous example, we used `$tasks` as the array.
- `ELEMENT` is the name of a variable which will hold, each time round the loop, the value of the current element in the array. In the previous example, we decided to name this variable `$task` , although we could have called it anything.
- `BLOCK` is a section of Puppet code. This could consist of a function call, resource declarations, include statements, conditional statements: anything which you can put in a Puppet manifest, you can also put inside a loop block. In the previous example, the only thing in the block was the `file` resource, which creates `/usr/local/bin/$task` .

Iterating over hashes

The `each` function works not only on arrays, but also on hashes. When iterating over a hash, the loop takes two `ELEMENT` parameters: the first is the hash key, and the second is the value. The following example shows how to use `each` to iterate over a hash resulting from a Facter query (`iteration_hash.pp`):

```
$nics = $facts['networking']['interfaces']
$nics.each | String $interface, Hash $attributes | {
  notice("Interface ${interface} has IP ${attributes['ip']}")
}
```

The list of interfaces returned by `$facts['networking']['interfaces']` is a hash, where the key is the name of the interface (for example, `lo0` for the local loopback interfaces) and the value is a hash of the interface's attributes (including the IP address, netmask, and so on). Applying the manifest in the previous example gives this result (on my Vagrant box):

```
sudo puppet apply /examples/iteration_hash.pp
Notice: Scope(Class[main]): Interface enp0s3 has IP 10.0.2.15
Notice: Scope(Class[main]): Interface lo has IP 127.0.0.1
```

Summary

In this lab, we've gained an understanding of how Puppet's variable and data type system works, including the basic data types: Strings, Numbers, Booleans, Arrays, and Hashes. We've seen how to interpolate variables in strings and how to quickly create sets of similar resources using an array of resource names. We've learned how to set common attributes for resources using a hash of attribute-value pairs and the attribute splat operator.

We've seen how to use variables and values in expressions, including arithmetic expressions, and explored the range of Puppet's comparison operators to generate Boolean expressions. We've used conditional expressions to build `if…else` and `case` statements and had a brief introduction to regular expressions.

We've learned how Puppet's Facter subsystem supplies information about the node via the facts hash and how to use facts in our own manifests and in expressions. We've pointed out some key facts, including the operating system release, the system memory capacity, and the system hostname. We've seen how to create custom external facts, such as a `cloud` fact, and how to dynamically generate fact information using executable facts.

Finally, we've learned about iteration in Puppet using the `each` function and how to create multiple resources based on data from arrays or hashes, including Facter queries.

In the next lab, we'll stay with the topic of data and explore Puppet's powerful Hiera database. We'll see what problems Hiera solves, look at how to set up and query Hiera, how to write data sources, how to create Puppet resources directly from Hiera data, and also how to use Hiera encryption to manage secret data.