Contents

Intro to viash

A step in the rendering of the video contains of one aspect that can be considered on its own. Understanding the logic of a step, however, is not sufficient as we have seen before. We also need to define the environment in which the step has to be performed.

In other words, we need to understand *what* needs to run and *how* it should run. `combine_plots`, for instance, takes as input a number of plots (`png` images) and combines them into a plot. As introduced earlier, we can use `ffmpeg` for this and it's then just a matter of getting the proper arguments for the tool right. That's basically what we did in the previous section. But it did not stop there, we had to explicitly install the tool in order to run it. viash allows to do exactly this: specify the *what* and the *how*.

Before actually porting the Civilization postgame scripts to viash, let's first look at some small examples to gradually demonstrate how viash works. Let's start by installing the latest release of viash!

Installing viash

Installation of viash is explained in here.

Since we want to keep this tutorial self-contained, we will download and install the latest (binary) release and install it locally. You'll need the following for htis:

- Access to a Linux, UNIX, Mac system or Windows with WSL(2)
- A terminal application with a Bash shell
- Java 8 or higher installed

You can install viash for your current user by downloading it and placing it in the 'bin' directory in your home folder.

```
mkdir -p ~/bin/
wget https://github.com/data-intuitive/viash/releases/download/v0.3.1/viash -qO ~/bin/viash
chmod +x ~/bin/viash
```

If viash is installed correctly, you should be able to invoke the help message by executing the following:

```
> viash -h
viash 0.3.2 (c) 2020 Data Intuitive

viash is a spec and a tool for defining execution contexts and converting execution instructions to

This program comes with ABSOLUTELY NO WARRANTY. This is free software, and you are welcome to redis
  https://github.com/data-intuitive/viash/blob/master/LICENSE.md

Usage:
  viash run config.vsh.yaml -- [arguments for script]
  viash build config.vsh.yaml
  viash test config.vsh.yaml
  viash ns build
  viash ns test

Check the help of a subcommand for more information, or the API available at:
  https://www.data-intuitive.com/viash_docs

Arguments:
  -h, --help      Show help message
  -v, --version   Show version of this program

Subcommands:
  run
  build
  test
  ns
```

Example 1: a minimal viash config file

A core concept in viash is the viash config, which is a YAML file containing information on software component, such as its parameters, its requirements, and some documentation.

Let us start with the smallest possible viash config, which is an almost trivial wrapper around `ls`. `ls` is a Unix command used to list all files in a directory.

`src/intro_example1.vsh.yaml`:

```yaml
functionality:
  name: intro_example1
  resources:
    - type: executable
      path: ls
```

`viash run` is a command for running a component as defined by the viash config. You can run it as follows:

```
> viash run src/intro_example1.vsh.yaml
Makefile
README.html
README.md
README.Rmd
src
```

Perhaps unsurprisingly, this performs an `ls` in the *current* directory which in this case is where viash is running. This example, while illustrative, does not capture what viash is and can be used for. It's just a wrapper around the `ls` command.

Let's go one step further.

Example 2: adding some arguments

Software components are (usually) not useful unless they have some arguments which you can specify and change.

`src/intro_example2.vsh.yaml`:

```yaml
functionality:
  name: intro_example2
  arguments:
    - name: "-l"
      type: boolean_true
    - name: "-a"
      type: boolean_true
  resources:
    - type: executable
      path: ls
```

We added two arguments to the `arguments` list. The arguments are flags and if we specify for `-l` it means *long listing* is one which corresponds to `boolean_true`.

This is what happens when you run viash in a few different scenarios:

```
> viash run src/intro_example2.vsh.yaml
Makefile
README.html
README.md
README.Rmd
src
```

There is no difference with the previous version of the component. Now, let us pass the argument `-l` to `intro_example2`:

```
> viash run src/intro_example2.vsh.yaml -- -l
total 120
-rw-rw-r--. 1 rcannood rcannood   665 Feb  4 16:40 Makefile
-rw-rw-r--. 1 rcannood rcannood 78273 Feb  4 06:11 README.html
```

```
-rw-rw-r--. 1 rcannood rcannood 22946 Feb  4 06:11 README.md
-rw-r--r--. 1 rcannood rcannood 10602 Feb  4 15:41 README.Rmd
drwxr-xr-x. 1 rcannood rcannood   294 Jan 28 08:35 src
```

Please note that options *before* the `--` are considered for viash while options after the `--` are for the tool that is wrapped (in this case `ls`).

Example 3: setting different argument types

Not all arguments are boolean flags such as specified in the previous example. In this viash config, we added an extra argument that corresponds to the path which we want to *list*.

`src/intro_example3.vsh.yaml`:

```yaml
functionality:
  name: intro_example3
  arguments:
    - name: "-l"
      type: boolean_true
    - name: "-a"
      type: boolean_true
    - name: "path"
      type: file
      default: .
  resources:
    - type: executable
      path: ls
```

Running this component will still list the contents of the current directory (like before).

```
> viash run src/intro_example3.vsh.yaml -- -l
total 120
-rw-rw-r--. 1 rcannood rcannood   665 Feb  4 16:40 Makefile
-rw-rw-r--. 1 rcannood rcannood 78273 Feb  4 06:11 README.html
-rw-rw-r--. 1 rcannood rcannood 22946 Feb  4 06:11 README.md
-rw-r--r--. 1 rcannood rcannood 10602 Feb  4 15:41 README.Rmd
drwxr-xr-x. 1 rcannood rcannood   294 Jan 28 08:35 src
```

However, we can now also list the contents of a different directory.

```
> viash run src/intro_example3.vsh.yaml -- src/ -l
total 28
-rw-r--r--. 1 rcannood rcannood  89 Jan 28 08:35 intro_example1.vsh.yaml
-rw-r--r--. 1 rcannood rcannood 186 Jan 28 08:35 intro_example2.vsh.yaml
-rw-r--r--. 1 rcannood rcannood 239 Jan 28 08:35 intro_example3.vsh.yaml
-rw-r--r--. 1 rcannood rcannood 543 Jan 28 08:35 intro_example4.vsh.yaml
-rw-r--r--. 1 rcannood rcannood 613 Jan 28 08:35 intro_example5.vsh.yaml
-rw-r--r--. 1 rcannood rcannood 593 Jan 28 08:35 intro_example6.vsh.yaml
-rw-r--r--. 1 rcannood rcannood  58 Jan 28 08:35 script.sh
```

You can always retrieve information about the component by requesting the included help.

```
> viash run src/intro_example3.vsh.yaml -- -h


Options:
    -l
        type: boolean_true

    -a
        type: boolean_true

    file
        type: file, default: .
```

Note that there are many more argument types than a flag or a file. These are not very useful for now, but come in handy when wrapping R/Python/JavaScript scripts. For more information, see the documentation regarding the functionality specifications.

Example 4: adding documentation

The help from the last `intro_example3` does not show a lot of useful information. Let's add some documentation regarding the component and its parameters.

`src/intro_example4.vsh.yaml`:

```yaml
functionality:
  name: intro_example4
  version: 0.4
  description: |
    List information about the files (the current directory by default)
    in alphabetical order.
  arguments:
    - name: "-l"
      type: boolean_true
      description: "Use a long listing format."
    - name: "-a"
      type: boolean_true
      description: "Do not ignore entries starting with '.'."
    - name: "path"
      type: file
      description: "Which directory to list the contents of."
      default: .
  resources:
    - type: executable
      path: ls
```

In doing so, the help message becomes a lot more useful in reminding

yourself and other users how to use the components.

```
> viash run src/intro_example4.vsh.yaml -- -h
List information about the files (the current directory by default)
in alphabetical order.

Options:
    -l
        type: boolean_true
        Use a long listing format.

    -a
        type: boolean_true
        Do not ignore entries starting with '.'.

    file
        type: file, default: .
        Which directory to list the contents of.
```

Example 4 part 2: building an *executable*

Suppose `intro_example4` from above is exactly what we need as standalone tool for ourselves or other people to use. Obviously, providing everyone access to viash and then letting them access the `intro_example4.vsh.yaml` file in order to run the above commands would not simplify things at all!

Time to introduce a second viash command, namely `viash build`. This command takes a viash config as input, and generates an executable as output.

```
> viash build src/intro_example4.vsh.yaml -o bin
```

After running the above command, viash will have generated a file at `bin/intro_example4`. It contains all the functionality that we saw in the above examples:

```
> bin/intro_example4 -h
List information about the files (the current directory by default)
in alphabetical order.

Options:
    -l
        type: boolean_true
        Use a long listing format.

    -a
        type: boolean_true
        Do not ignore entries starting with '.'.

    file
```

```
        type: file, default: .
        Which directory to list the contents of.
```

```
> bin/intro_example4 src/ -l
total 28
-rw-r--r--. 1 rcannood rcannood  89 Jan 28 08:35 intro_example1.vsh.yaml
-rw-r--r--. 1 rcannood rcannood 186 Jan 28 08:35 intro_example2.vsh.yaml
-rw-r--r--. 1 rcannood rcannood 239 Jan 28 08:35 intro_example3.vsh.yaml
-rw-r--r--. 1 rcannood rcannood 543 Jan 28 08:35 intro_example4.vsh.yaml
-rw-r--r--. 1 rcannood rcannood 613 Jan 28 08:35 intro_example5.vsh.yaml
-rw-r--r--. 1 rcannood rcannood 593 Jan 28 08:35 intro_example6.vsh.yaml
-rw-r--r--. 1 rcannood rcannood  58 Jan 28 08:35 script.sh
```

You can now share this `bin/intro_example4` file with others, or add it to your `~/bin` directory to turn it into a system-wide command.

Example 5: running the component inside a Docker container

In the above examples, we ran the components on our local system. This is simple as long as the wrapped tool at hand (`ls` in this case) is always available on the local system. However, this assumption generally does not hold true. viash not only supports running components on the native system, but can also run components inside a Docker container.

To make use of this functionality, we need to get into the the 'platforms' section of the viash config, which can contain one or more execution platforms. In this case, we defined platforms: a *native* one (local machine) and a *docker* one.

`src/intro_example5.vsh.yaml`:

```yaml
functionality:
  name: intro_example5
  version: 0.5
  description: |
    List information about the files (the current directory by default)
    in alphabetical order.
  arguments:
    - name: "-l"
      type: boolean_true
      description: "Use a long listing format."
    - name: "-a"
      type: boolean_true
      description: "Do not ignore entries starting with '.'."
    - name: "path"
      type: file
      description: "Which directory to list the contents of."
      default: .
  resources:
```

```
    - type: executable
      path: ls
platforms:
  - type: native
  - type: docker
    image: ubuntu:latest
```

By default, viash will use the first platform specified in the viash config, which in this case the native platform. In order to build an executable which uses Docker in the backend, we need to pass this information as follows:

```
> viash build src/intro_example5.vsh.yaml -o bin -p docker
```

The executable `bin/intro_example5` now automatically runs inside Docker.

```
> bin/intro_example5 / -l
total 20
lrwxrwxrwx.   1 root root    7 Jul 27  2020 bin -> usr/bin
dr-xr-xr-x.   7 root root 4096 Feb  3 13:26 boot
drwxr-xr-x.  24 root root 5200 Feb  4 12:12 dev
drwxr-xr-x.   1 root root 5336 Feb  4 15:58 etc
drwxr-xr-x.   1 root root   16 Dec 13 05:08 home
lrwxrwxrwx.   1 root root    7 Jul 27  2020 lib -> usr/lib
lrwxrwxrwx.   1 root root    9 Jul 27  2020 lib64 -> usr/lib64
drwx------.   1 root root    0 Oct 19 23:33 lost+found
drwxr-xr-x.   1 root root    0 Jul 27  2020 media
drwxr-xr-x.   1 root root   16 Jan  4 12:13 mnt
drwxr-xr-x.   1 root root  188 Jan 20 09:58 opt
dr-xr-xr-x. 703 root root    0 Feb  3 14:15 proc
dr-xr-x---.   1 root root  270 Feb  3 13:24 root
drwxr-xr-x.  55 root root 1500 Feb  4 15:58 run
lrwxrwxrwx.   1 root root    8 Jul 27  2020 sbin -> usr/sbin
drwxr-xr-x.   1 root root    0 Jul 27  2020 srv
dr-xr-xr-x.  13 root root    0 Feb  3 13:15 sys
drwxrwxrwt.  59 root root 1640 Feb  4 16:04 tmp
drwxr-xr-x.   1 root root  106 Jan 31 05:50 usr
drwxr-xr-x.   1 root root  208 Jan  7 14:11 var
```

If the `ubuntu` image is not yet available on your system, this command will automatically fetch it before running the tool. You can verify for yourself that the result of this listing is not the same as what you would have if you ran on your local system.

Please note that if you wanted to do this exact thing by using Docker itself, you would have to use a CLI instruction like

```
> docker run --rm -v /:/mount ubuntu:latest ls /mount/ -l
total 20
lrwxrwxrwx.   1 root root    7 Jul 27  2020 bin -> usr/bin
dr-xr-xr-x.   7 root root 4096 Feb  3 13:26 boot
drwxr-xr-x.  24 root root 5200 Feb  4 12:12 dev
```

```
drwxr-xr-x.   1 root root 5336 Feb  4 15:58 etc
drwxr-xr-x.   1 root root   16 Dec 13 05:08 home
lrwxrwxrwx.   1 root root    7 Jul 27  2020 lib -> usr/lib
lrwxrwxrwx.   1 root root    9 Jul 27  2020 lib64 -> usr/lib64
drwx------.   1 root root    0 Oct 19 23:33 lost+found
drwxr-xr-x.   1 root root    0 Jul 27  2020 media
drwxr-xr-x.   1 root root   16 Jan  4 12:13 mnt
drwxr-xr-x.   1 root root  188 Jan 20 09:58 opt
dr-xr-xr-x. 720 root root    0 Feb  3 14:15 proc
dr-xr-x---.   1 root root  270 Feb  3 13:24 root
drwxr-xr-x.  55 root root 1500 Feb  4 15:58 run
lrwxrwxrwx.   1 root root    8 Jul 27  2020 sbin -> usr/sbin
drwxr-xr-x.   1 root root    0 Jul 27  2020 srv
dr-xr-xr-x.  13 root root    0 Feb  3 13:15 sys
drwxrwxrwt.  59 root root 1640 Feb  4 16:04 tmp
drwxr-xr-x.   1 root root  106 Jan 31 05:50 usr
drwxr-xr-x.   1 root root  208 Jan  7 14:11 var
```

While this is all still manageable, it could quickly become more complicated, but that is for a later section. In what follows, we will also come back not only to running inside a container but also generating a container (based on a base image), tagging and versioning.

Example 6: Wrapping a script

While running a command wrapped as a viash component could be useful in *some* form or another, we will usually want to run something a bit more custom or elaborate. Say you want to run the `intro_example5` component from above but this time filtering out certain files/directories based on their name. We could do just that by means of a simple CLI instruction that we put in a script:

`src/script.sh`:

```
#!/bin/bash

eval "ls \"$par_path\" | grep '$par_filter'"
```

In combination with the following viash config:

`src/intro_example6.vsh.yaml`:

```
functionality:
  name: intro_example6
  version: 0.6
  description: |
    List information about the files (the current directory by default)
    in alphabetical order, filtered by a regular expression.
  arguments:
    - name: "path"
      type: file
```

```
      description: "Which directory to list the contents of."
      default: .
    - name: "--filter"
      type: string
      description: "A regular expression to filter the listed files."
      default: '.*'
  resources:
    - type: bash_script
      path: script.sh
platforms:
  - type: native
  - type: docker
    image: ubuntu:latest
```

We get results like this:

```
> viash run src/intro_example6.vsh.yaml -p docker -- /etc --filter "^h.*"
host.conf
hostname
hosts
hp
httpd
```

A lot is happening here at once, so let's unwrap this. We did not *build* the executable in this example, but just run `viash run` on on the viash config. This config contains a pointer (relative path) to the `script.sh` file that contains parameters. Those parameters are defined in the viash config and are automatically resolved and parsed when running the wrapped viash version of the script. The `docker` platform is defined in the viash config as well, so we can just run it inside the respective container. The `--filter` argument takes a regular expression, it is simply passed to `grep` in `script.sh`.

Please note that when we decide to *build* a `intro_example6` executable (for a specific platform), again this executable is self-contained. It includes the necessary Docker information, command line parsing logic and the script itself. So there is not need for additional customization.

If you would want to achieve something similar with just Docker without viash, you are in for some serious Bash development. But it does not stop here, because in addition to support for wrapping Bash scripts, viash also supports wrapping Python, R, JavaScript, and Scala scripts.