Arca

Release 0.3.3

Dec 10, 2019

Contents

Arca is a library for running Python functions (callables) from git repositories in various states of isolation. Arca can also cache the results of these callables using dogpile.cache.

CHAPTER 1

Quickstart

1.1 Glossary

- **Arca** - name of the library. When written as *Arca*, the main interface class is being referenced.
- **Task** - definition of the function (callable), consists of a reference to the object and arguments.
- **Backend** - a way of running tasks.

1.2 Example

To run a Hello World example you'll only need the *arca.Arca* and *arca.Task* classes. *Task* is used for defining the task that's supposed to be run in the repositories. *Arca* takes care of all the settings and provides the basic API for running the tasks.

Let's say we have the following file, called `hello_world.py`, in a repository `https://example.com/hello_word.git`, on branch `master`.

```
def say_hello():
   return "Hello World!"
```

To call the function using Arca, the following example would do so:

```
from arca import Arca, Task

task = Task("hello_world:say_hello")
arca = Arca()

result = arca.run("https://example.com/hello_word.git", "master", task)
print(result.output)
```

The code would print `Hello World!`.

`result` would be a *`Result`* instance. *`Result`* has three attributes, `output` with the return value of the function call, `stdout` and `stderr` contain things printed to the standard outputs (see the section about *Result* for more info about the capture of the standard outputs). If the task fails, *`arca.exceptions.BuildError`* would be raised.

By default, the *Current Environment Backend* is used to run tasks, which uses the current Python, launching the code in a subprocess. You can learn about backends *here*.

CHAPTER 2

Installation

2.1 Requirements

- Python >= 3.6

Requirements for certain backends:

- Pipenv (for certain usecases in *Virtualenv Backend*)
- Docker (for *Docker Backend* and *Vagrant Backend*)
- Vagrant (for the *Vagrant Backend*)

2.2 Installation

To install the last stable version:

```
python -m pip install arca
```

If you want to use the Docker backend:

```
python -m  pip install arca[docker]
```

Or if you want to use the Vagrant backend:

```
python -m pip install arca[vagrant]
```

Or if you wish to install the upstream version:

```
python -m pip install git+https://github.com/pyvec/arca.git#egg=arca
python -m pip install git+https://github.com/pyvec/arca.git#egg=arca[docker]
python -m pip install git+https://github.com/pyvec/arca.git#egg=arca[vagrant]
```

CHAPTER 3

Settings

3.1 Configuring Arca

There are multiple ways to configure *Arca* and its backends. (The used options are described *bellow*.)

1. You can initialize the class and backends directly and set it's options via constructor arguments.

```
from arca import Arca, VenvBackend

arca = Arca(
  base_dir=".custom_arca_dir",
  backend=VenvBackend(cwd="utils")
)
```

This option is the most direct but it has one caveat - options set by this method cannot be overridden by the following methods.

2. You can pass a dict with settings. The keys have to be uppercase and prefixed with `ARCA_`. Keys for backends can be set in two ways. The first is generic `ARCA_BACKEND_<key>`, the second has a bigger priority `ARCA_<backend_name>_BACKEND_<key>`. For example the same setting as above would be written as:

```
arca = Arca(settings={
  "ARCA_BASE_DIR": ".custom_arca_dir",
  "ARCA_BACKEND": "arca.VenvBackend",
  "ARCA_VENV_BACKEND_CWD": "utils",
  "ARCA_BACKEND_CWD": "",  # this one is ignored since it has lower priority
})
```

3. You can configure *Arca* with environ variables, with keys being the same as in the second method. Environ variables override settings from the second method.

You can combine these methods as long as you remember that options explicitly specified in constructors cannot be overridden by the settings and environ methods.

3.2 Basic options

This section only describes basic settings, visit the *cookbook* for more.

3.2.1 Arca class

base_dir (*ARCA_BASE_DIR*)

Arca needs to clone repositories and for certain backends also store some other files. This options determines where the files should be stored. The default is `.arca`. If the folder doesn't exist it's created.

backend (*ARCA_BACKEND*)

This options tells how the tasks should be launched. This setting can be provided as a string, a class or a instance. The default is `arca.CurrentEnvironmentBackend`, the *Current Environment Backend*.

3.2.2 Backends

This section describes settings that are common for all the backends.

requirements_location (*ARCA_BACKEND_REQUIREMENTS_LOCATION*)

Tells backends where to look for a requirements file in the repositories, so it must be a relative path. You can set it to `None` to indicate that requirement file should be ignored. The default is `requirements.txt`. If the path file doesn't exist in the repository than no requirements are installed.

requirements_timeout (*ARCA_BACKEND_REQUIREMENTS_TIMEOUT*)

Tells backends how long the installing of requirements can take, in seconds. The default is 120 seconds. If the limit is exceeded `BuildTimeoutError` is raised.

pipfile_location (*ARCA_BACKEND_PIPFILE_LOCATION*)

Tells backends where to look for `Pipfile` and `Pipfile.lock` files, for Pipenv. It must be a relative path to a directory. You can set it to `None` to indicate that Pipenv files should be ignored. The default is an empty string, the root of the repository. If there are Pipenv files in the repository alongside a requirement file than Pipenv takes precedence.

Both `Pipfile` and `Pipfile.lock` must be present in the repository for Pipenv to be used. If only one of them is present then an exception is raised. The `--deploy` flag is used, meaning that the `Pipfile.lock` must be up to date with `Pipfile`.

cwd (*ARCA_BACKEND_CWD*)

Tells Arca in what working directory the tasks should be launched, so again a relative path. The default is the root of the repository.

CHAPTER 4

Tasks

arca.Task instances are used to define what should be run in the repositories. The definition consists of a string representation of a callable and arguments.

EntryPoints (from the entrypoints library) are used for defining the callables. Any callable can be used if the result is json-serializable by the standard library `json`.

Let's say we have file `package/test.py` in the repository:

```
def func():
  x = Test()
  return x.run()

class Test:
  def run(self):
    ...
    return "Hello!"

  @staticmethod
  def method():
    x = Test()
    return x.run()
```

In that case, the following two tasks would have the same result:

```
task1 = Task("package.test:func")
task2 = Task("package.test:Test.method")
```

4.1 Arguments

Both positional and keyword arguments can be provided to the task, however they need to be json-serializable (so types `dict`, `list`, `str`, `int`, `float`, `bool` or `None`).

Let's say we the following file `test.py` in the repository:

```
def func(x, *, y=5):
  return x * y
```

The following tasks would use the fuction (with the default `y`):

```
task1 = Task("test:func", args=[5])  # -> result would be 25
task2 = Task("test:func", kwargs={"x": 5})  # -> result would be 25 again
```

Since the `x` parameter is positional, both ways can be used. However, if we wanted to set `y`, the task would be set up like this:

```
task1 = Task("test:func", args=[5], kwargs={"y": 10})  # -> 50
task2 = Task("test:func", kwargs={"x": 5, "y": 10})  # -> 50 again
```

4.2 Timeout

The *arca.Task* class allows for a timeout to be defined with the task with the keyword argument `timeout`. It must be a positive integer. The default value is 5 seconds.

When a task exceeds a timeout, *arca.exceptions.BuildTimeoutError* is raised.

4.3 Result

The output of a task is stored and returned in a *arca.Result* instance. Anything that's json-serializable can be returned from the entrypoints. The *arca.Result* instances contain three attributes. `output` contains the value returned from the entrypoint. `stdout` and `stderr` contain things written to the standard outputs.

Arca uses `contextlib.redirect_stdout()` and `contextlib.redirect_stderrr()` to catch the standard outputs, which only redirect things written from standard Python code – for example output from a subprocess is not caught. Due to the way backends launch tasks the callables cannot output anything that is not redirectable by these two context managers.

CHAPTER 5

Caching

Arca can cache results of the tasks using dogpile.cache. The default cache backend is `dogpile.cache.null`, so no caching.

You can setup caching with the backend setting `ARCA_CACHE_BACKEND` and all the arguments needed for setup can be set using `ARCA_CACHE_BACKEND_ARGUMENTS` which can either be a dict or a json string.

Example setup:

```
arca = Arca(settings={
    "ARCA_CACHE_BACKEND": "dogpile.cache.redis",
    "ARCA_CACHE_BACKEND_ARGUMENTS": {
        "host": "localhost",
        "port": 6379,
        "db": 0,
    }
})
```

To see all available cache backends and their settings, please visit the `dogpile.cache` documentation. Some of the other backends might have other python dependencies.

When *`Arca`* is being initialized, a check is made if the cache backend is writable and readable, which raises an *`arca.exceptions.ArcaMisconfigured`* if it's not. If the cache requires some python dependency `ModuleNotFoundError` will be raised. If you wish to ignore these errors, `ignore_cache_errors` setting can be used.

CHAPTER 6

Backends

There are currently four different backends. They can also be initialized in few different ways, consistent with general settings. You can use the `ARCA_BACKEND` setting or you can pass a `backend` keyword directly to *Arca*.

The backend setting can be either a string, class or an instance. All the initializations shown bellow are equivalent, but again, as mentioned in *Configuring Arca*, the `backend` keyword cannot be overridden by settings or environ variables.

```
from arca import Arca, DockerBackend

Arca(settings={"ARCA_BACKEND": "arca.backend.DockerBackend"})
Arca(settings={"ARCA_BACKEND": DockerBackend})
Arca(backend="arca.backend.DockerBackend")
Arca(backend=DockerBackend)
Arca(backend=DockerBackend())
```

Setting up backends is based on the same principle as setting up *Arca*. You can either pass keyword arguments when initializing the backend class or you can use settings (described in more details in *Configuring Arca*). For example these two calls are equivalent:

```
from arca import Arca, DockerBackend

Arca(settings={
  "ARCA_BACKEND": "arca.backend.DockerBackend",
  "ARCA_BACKEND_PYTHON_VERSION": "3.6.4"
})
Arca(backend=DockerBackend(python_version="3.6.4"))
```

As mentioned in *Basic options*, there are two options common for all backends. (See that section for more details.)

- **requirements_location**
- **requirements_timeout**
- **pipfile_location**
- **cwd**

6.1 Current Environment

arca.backend.CurrentEnvironmentBackend

This backend is the default option, it runs the tasks with the same Python that's used to run Arca, in a subprocess. There are no extra settings for this backend. All the requirements in repositories are ignored completely.

(possible settings prefixes: `ARCA_CURRENT_ENVIRONMENT_BACKEND_` and `ARCA_BACKEND_`)

6.2 Virtual Environment

arca.backend.VenvBackend

This backend uses the Python virtual environments to run the tasks. The environments are created from the Python used to run Arca and they are shared between repositories that have the same exact requirement file. The virtual environments are stored in folder `venv` in folder determined by the *Arca* `base_dir` setting, usually `.arca`.

For installing requirements using Pipenv it must be available to be launched by the current user. Disabling Pipenv can be done by setting the **pipfile_location** to `None`.

(possible settings prefixes: `ARCA_VENV_BACKEND_` and `ARCA_BACKEND_`)

6.3 Docker

arca.backend.DockerBackend

This backend runs tasks in docker containers. To use this backend the user running Arca needs to be able to interact with `docker` (see documentation).

This backend first creates an image with requirements and dependencies installed so the installation only runs once. By default the images are based on custom images, which have Python and several build tools pre-installed. These images are based on `debian` (slim `stretch` version) and use pyenv to install Python. You can specify you want to base your images on a different image with the `inherit_image` setting.

Once Arca has an image with the requirements installed, it launches a container for each task and kills it when the task finishes. This can be modify by setting `keep_container_running` to `True`, then the container is not killed and can be used by different tasks running from the same repository, branch and commit. This can save time on starting up containers before each task. You can then kill the containers by calling `DockerBackend` method `stop_containers`.

If you're using Arca on a CI/CD tool or somewhere docker images are not kept long-term, you can setup pushing images with the installed requirements and dependencies to a docker registry and they will be pulled next time instead of building them each time. It's set using `use_registry_name` and you'll have to be logged in to docker using `docker login`. If you can't use `docker login` (for example in PRs on Travis CI), you can set `registry_pull_only` and Arca will only attempt to pull from the registry and not push new images.

Settings:

- **python_version**: What Python version should be used. In theory any of these versions could be used, but only CPython 3.6 has been tested. The default is the Python version of the current environment. This setting is ignored if `inherit_image` is set.
- **keep_container_running**: When `True`, containers aren't killed once the task finishes. Default is `False`.
- **apt_dependencies**: For some python libraries, system dependencies are required, for example `libxml2-dev` and `libxslt-dev` are needed for `lxml`. With this settings you can specify a list of system dependencies

that will be installed via debian `apt-get`. This setting is ignored if `inherit_image` is set since Arca can't determined how to install requirements on an unknown system.

- **disable_pull**: Disable pulling prebuilt Arca images from Docker Hub and build even the base images locally.
- **inherit_image**: If you don't wish to use the Arca images you can specify what image should be used instead. Pipenv must be available in the image if the repositories contain Pipenv files. Alternativelly Pipenv can be disabled by setting the option **pipfile_location** to `None`.
- **use_registry_name**: Uses this registry to store images with installed requirements and dependencies to, tries to pull image from the registry before building it locally to save time.
- **registry_pull_only**: Disables pushing to registry.

(possible settings prefixes: `ARCA_DOCKER_BACKEND_` and `ARCA_BACKEND_`)

6.4 Vagrant

arca.backend.VagrantBackend

If you're extra paranoid you can use Vagrant to completely isolate the runtime in a Virtual Machine (VM). This backend is actually a subclass of `DockerBackend` and uses docker in the VM to run the tasks. Docker and Vagrant must be runnable by the current user.

The backend works by building the image with requirements and dependencies locally and pushing it to registry using `use_to_registry_name`. Then a VM is launched and the image is pulled there from the registry. This takes some time when first launching the VM, but if the VM is reused often, the upload/download time is shorted. The built images are also not lost when the VM is destroyed.

The backend inherits all the settings of `DockerBackend` (**keep_containers_running** is `True` by default) and has these extra settings:

- **box**: Vagrant box used in the VM. Either has to have docker version >= 1.8 or not have docker at all, in which case it will be installed when spinning up the VM. The default is ailispaw/barge.
- **provider**: Vagrant provider, default is `virtualbox`. Visit vagrant docs for more.
- **quiet**: Tells Vagrant and Fabric (which is used to run the task in the VM) to be quiet. Default is `True`. Vagrant and Docker output is logged in separate files for each run in a folder `logs` in the *Arca* `base_dir`. The filename is logged in the arca logger (see bellow)
- **keep_vm_running**: Should the VM be kept up once a task finishes? By default `False`. If set to `True`, *stop_vm* can be used to stop the VM.
- **destroy**: When stopping the VM (either after a task or after `stop_vm()` is called), should the VM be destroyed (= deleted) or just halted? `False` by default.

(possible settings prefixes: `ARCA_VAGRANT_BACKEND_` and `ARCA_BACKEND_`)

6.5 Your own

You can also create your own backend and pass it to *Arca*. It has be a subclass of *arca.BaseBackend* and it has to implement its *run* method.

CHAPTER 7

Cookbook

7.1 I need some files from the repositories

Besides running functions from the repositories, there also might be some files in the repositories that you need, e.g. images for a webpage. With the *Arca* method *static_filename* you can get the absolute path to that file, to where Arca cloned it. The method accepts a relative path (can be a pathlib.Path or str) to the file in the repository.

Example call (file images/example.png from the branch master of https://example.com/hello_word.git):

```
arca = Arca()

path_to_file = arca.static_filename("https://example.com/hello_word.git",
                                    "master",
                                    "images/example.png")
```

path_to_file will be an absolute Path.

If the file is not in the repository, FileNotFoundError will be raised. If the provided relative path leads out of the repo, *FileOutOfRangeError* will be raised.

7.2 I will be running a lot of tasks from the same repository

Similarly as above, while you're building a webpage you might need to run a lot of tasks from the same repositories, to render all the individual pages. However Arca has some overhead for each launched task, but these two options can speed things up:

7.2.1 Singe pull

This option ensures that each branch is only cloned/pulled once per initialization of *Arca*. You can set it up with the *Arca* `single_pull` option (`ARCA_SINGLE_PULL` setting). This doesn't help to speedup the first task from a repository, however each subsequent will run faster. This setting is quite useful for keeping consistency, since the state of the repository can't change in the middle of running multiple tasks.

You can tell *Arca* to pull again (if a task from that repo/branch is called again) by calling the method *Arca.pull_again*:

```
arca = Arca()

...

# only this specific branch will be pulled again
arca.pull_again(repo="https://example.com/hello_word.git", branch="master")

# all branches from this repo will be pulled again
arca.pull_again(repo="https://example.com/hello_word.git")

# everything will be pulled again
arca.pull_again()
```

7.2.2 Running container

If you're using the *Docker* backend, you can speed up things by keeping the containers for running the tasks running. Since a container for each repository is launched, this can speed up things considerably, because starting up, copyting files and shutting down containers takes time.

This can be enabled by setting the `keep_container_running` option to `True`. When you're done with running the tasks you can kill the containers by calling the method *DockerBackend.stop_containers*:

```
arca = Arca(backend=DockerBackend())

...

arca.backend.stop_containers()
```

CHAPTER 8

Reference

8.1 Arca

class arca.**Arca**(*backend: Union[Callable, arca.backend.base.BaseBackend, str, arca.utils.NotSet] = NOT_SET, settings=None, single_pull=NOT_SET, base_dir=NOT_SET, ignore_cache_errors=NOT_SET*)

Basic interface for communicating with the library, most basic operations should be possible from this class.

Available settings:

- **base_dir**: Directory where cloned repositories and other files are stored (default: `.arca`)
- **single_pull**: Clone/pull each repository only once per initialization (default: `False`)
- **ignore_cache_errors**: Ignore all cache error initialization errors (default: `False`)

cache_key(*repo: str, branch: str, task: arca.task.Task, git_repo: git.repo.base.Repo*) → str

Returns the key used for storing results in cache.

current_git_hash(*repo: str, branch: str, git_repo: git.repo.base.Repo, short: bool = False*) → str

Parameters

- **repo** – Repo URL
- **branch** – Branch name
- **git_repo** – `Repo` instance.
- **short** – Should the short version be returned?

Returns Commit hash of the currently pulled version for the specified repo/branch

get_backend_instance(*backend: Union[Callable, arca.backend.base.BaseBackend, str, arca.utils.NotSet]*) → arca.backend.base.BaseBackend

Returns a backend instance, either from the argument or from the settings.

Raises ***ArcaMisconfigured*** – If the instance is not a subclass of `BaseBackend`

get_files(*repo: str*, *branch: str*, ***, *depth: Optional[int] = 1*, *reference: Optional[pathlib.Path] = None*) → Tuple[git.repo.base.Repo, pathlib.Path]

Either clones the repository if it's not cloned already or pulls from origin. If `single_pull` is enabled, only pulls if the repo/branch combination wasn't pulled again by this instance.

Parameters

- **repo** – Repo URL
- **branch** – Branch name
- **depth** – See *run()*
- **reference** – See *run()*

Returns A `Repo` instance for the repo and a `Path` to the location where the repo is stored.

get_path_to_repo(*repo: str*) → pathlib.Path

Returns a `Path` to the location where all the branches from this repo are stored.

Parameters **repo** – Repo URL

Returns Path to where branches from this repository are cloned.

get_path_to_repo_and_branch(*repo: str*, *branch: str*) → pathlib.Path

Returns a `Path` to where this specific branch is stored on disk.

Parameters

- **repo** – Repo URL
- **branch** – branch

Returns Path to where the specific branch from this repo is being cloned.

get_reference_repository(*reference: Optional[pathlib.Path]*, *repo: str*) → Optional[pathlib.Path]

Returns a repository to use in clone command, if there is one to be referenced. Either provided by the user of generated from already cloned branches (master is preferred).

Parameters

- **reference** – Path to a local repository provided by the user or None.
- **repo** – Reference for which remote repository.

get_repo(*repo: str*, *branch: str*, ***, *depth: Optional[int] = 1*, *reference: Optional[pathlib.Path] = None*) → git.repo.base.Repo

Returns a `Repo` instance for the branch.

See *run()* for arguments descriptions.

make_region() → dogpile.cache.region.CacheRegion

Returns a `CacheRegion` based on settings.

- Firstly, a backend is selected. The default is `NullBackend <dogpile.cache.backends.null.NullBackend>`.
- Secondly, arguments for the backends are generated. The arguments can be passed as a dict to the setting or as a json string. If the arguments aren't a dict or aren't convertible to a dict, `ArcaMisconfigured` is raised.
- Lastly, the cache is tested if it works

All errors can be suppressed by the `ignore_cache_errors` setting.

Raises

- **ModuleNotFoundError** – In case dogpile has trouble importing the library needed for a backend.
- ***ArcaMisconfigured*** – In case the cache is misconfigured in any way or the cache doesn't work.

pull_again(*repo: Optional[str] = None*, *branch: Optional[str] = None*) → None
When `single_pull` is enables, tells Arca to pull again.

If `repo` and `branch` are not specified, pull again everything.

Parameters

- **repo** – (Optional) Pull again all branches from a specified repository.
- **branch** – (Optional) When `repo` is specified, pull again only this branch from that repository.

Raises **ValueError** – If `branch` is specified and `repo` is not.

repo_id(*repo: str*) → str
Returns an unique identifier from a repo URL for the folder the repo is gonna be pulled in.

run(*repo: str*, *branch: str*, *task: arca.task.Task*, ***, *depth: Optional[int] = 1*, *reference: Union[pathlib.Path, str, None] = None*) → arca.result.Result
Runs the `task` using the configured backend.

Parameters

- **repo** – Target git repository
- **branch** – Target git branch
- **task** – Task which will be run in the target repository
- **depth** – How many commits back should the repo be cloned in case the target repository isn't cloned yet. Defaults to 1, must be bigger than 0. No limit will be used if `None` is set.
- **reference** – A path to a repository from which the target repository is forked, to save bandwidth, *–dissociate* is used if set.

Returns A *Result* instance with the output of the task.

Raises

- ***PullError*** – If the repository can't be cloned or pulled
- ***BuildError*** – If the task fails.

save_hash(*repo: str*, *branch: str*, *git_repo: git.repo.base.Repo*)
If `single_pull` is enabled, saves the current git hash of the specified repository/branch combination, to indicate that it shouldn't be pull again.

should_cache_fn(*value: arca.result.Result*) → bool
Returns if the result `value` should be cached. By default, always returns `True`, can be overriden.

static_filename(*repo: str, branch: str, relative_path: Union[str, pathlib.Path], *, depth: Optional[int] = 1, reference: Union[pathlib.Path, str, None] = None*) → pathlib.Path
Returns an absolute path to where a file from the repo was cloned to.

Parameters

- **repo** – Repo URL
- **branch** – Branch name
- **relative_path** – Relative path to the requested file

- **depth** – See *run()*
- **reference** – See *run()*

Returns Absolute path to the file in the target repository

Raises

- ***FileOutOfRangeError*** – If the relative path leads out of the repository path
- **FileNotFoundError** – If the file doesn't exist in the repository.

validate_depth(*depth: Optional[int]*) → Optional[int]
Converts the depth to int and validates that the value can be used.

Raises **ValueError** – If the provided depth is not valid

validate_reference(*reference: Union[pathlib.Path, str, None]*) → Optional[pathlib.Path]
Converts reference to `Path`

Raises **ValueError** – If `reference` can't be converted to `Path`.

validate_repo_url(*repo: str*)
Validates repo URL - if it's a valid git URL and if Arca can handle that type of repo URL

Raises **ValueError** – If the URL is not valid

8.2 Task

class arca.**Task**(*entry_point: str*, *, *timeout: int = 5*, *args: Optional[Iterable[Any]] = None*, *kwargs: Optional[Dict[str, Any]] = None*)
A class for defining tasks the run in the repositories. The task is defined by an entry point, timeout (5 seconds by default), arguments and keyword arguments. The class uses `entrypoints.EntryPoint` to load the callables. As apposed to `EntryPoint`, only objects are allowed, not modules.

Let's presume we have this function in a package `library.module`:

```
def ret_argument(value="Value"):
    return value
```

This Task would return the default value:

```
>>> Task("library.module:ret_argument")
```

These two Tasks would returned an overridden value:

```
>>> Task("library.module:ret_argument", args=["Overridden value"])
>>> Task("library.module:ret_argument", kwargs={"value": "Overridden value"})
```

hash
Returns a SHA1 hash of the Task for usage in cache keys.

8.3 Result

class arca.**Result**(*result: Union[str, Dict[str, Any]]*)
For storing results of the tasks. So far only has one attribute, *output*.

output = None
The output of the task

stderr = None
What the function wrote to stderr

stdout = None
What the function wrote to stdout

8.4 Backends

8.4.1 Abstract classes

class arca.**BaseBackend**(***settings*)
Bases: object

Abstract class for all the backends, implements some basic functionality.

Available settings:

- **requirements_location**: Relative path to the requirements file in the target repositories. Setting to `None` makes Arca ignore requirements. (default is `requirements.txt`)
- **requirements_timeout**: The maximum time in seconds allowed for installing requirements. (default is 5 minutes, 300 seconds)
- **pipfile_location**: The folder containing `Pipfile` and `Pipfile.lock`. Pipenv files take precedence over requirements files. Setting to `None` makes Arca ignore Pipenv files. (default is the root of the repository)
- **cwd**: Relative path to the required working directory. (default is `""`, the root of the repo)

get_requirements_information(*path: pathlib.Path*) → Tuple[arca.backend.base.RequirementsOptions, Optional[str]]
Returns the information needed to install requirements for a repository - what kind is used and the hash of contents of the defining file.

get_setting(*key*, *default=NOT_SET*)
Gets a setting for the key.

Raises **KeyError** – If the key is not set and default isn't provided.

get_settings_keys(*key*)
Parameters can be set through two settings keys, by a specific setting (eg. `ARCA_DOCKER_BACKEND_KEY`) or a general `ARCA_BACKEND_KEY`. This function returns the two keys that can be used for this setting.

static hash_file_contents(*requirements_option: arca.backend.base.RequirementsOptions*, *path: pathlib.Path*) → str
Returns a SHA256 hash of the contents of `path` combined with the Arca version.

inject_arca(*arca*)
After backend is set for a *Arca* instance, the instance is injected to the backend, so settings can be accessed, files accessed etc. Also runs settings validation of the backend.

run(*repo: str*, *branch: str*, *task: arca.task.Task*, *git_repo: git.repo.base.Repo*, *repo_path: pathlib.Path*) → arca.result.Result
Executes the script and returns the result.

Must be implemented by subclasses.

Parameters

- **repo** – Repo URL
- **branch** – Branch name
- **task** – The requested *Task*
- **git_repo** – A Repo of the repo/branch
- **repo_path** – Path to the location where the repo is stored.

Returns The output of the task in a *Result* instance.

serialized_task(*task: arca.task.Task*) → Tuple[str, str]
Returns the name of the task definition file and its contents.

snake_case_backend_name
CamelCase -> camel_case

class arca.backend.base.**BaseRunInSubprocessBackend**(***settings*)
Bases: arca.backend.base.BaseBackend

Abstract class for backends which run scripts in subprocess.

get_or_create_environment(*repo: str*, *branch: str*, *git_repo: git.repo.base.Repo*, *repo_path: pathlib.Path*) → str
Abstract method which must be implemented in subclasses, which must return a str path to a Python executable which will be used to run the script.

See *BaseBackend.run* to see arguments description.

run(*repo: str*, *branch: str*, *task: arca.task.Task*, *git_repo: git.repo.base.Repo*, *repo_path: pathlib.Path*) → arca.result.Result
Gets a path to a Python executable by calling the abstract method get_image_for_repo and runs the task using subprocess.Popen

See *BaseBackend.run* to see arguments description.

8.4.2 Current environment

class arca.**CurrentEnvironmentBackend**(***settings*)
Bases: *arca.backend.base.BaseRunInSubprocessBackend*

Uses the current Python to run the tasks, however they're launched in a subprocess.

The requirements of the repository are completely ignored.

get_or_create_environment(*repo: str*, *branch: str*, *git_repo: git.repo.base.Repo*, *repo_path: pathlib.Path*) → str
Returns the path to the current Python executable.

8.4.3 Python virtual environment

class arca.**VenvBackend**(***settings*)
Bases: *arca.backend.base.BaseRunInSubprocessBackend*

Uses Python virtual environments (see venv), the tasks are then launched in a subprocess. The virtual environments are shared across repositories when they have the exact same requirements. If the target repository doesn't have requirements, it also uses a virtual environment, but just with no extra packages installed.

There are no extra settings for this backend.

get_or_create_environment(*repo: str*, *branch: str*, *git_repo: git.repo.base.Repo*, *repo_path: pathlib.Path*) → str
Handles the requirements in the target repository, returns a path to a executable of the virtualenv.

get_or_create_venv(*path: pathlib.Path*) → pathlib.Path
Gets the location of the virtualenv from *get_virtualenv_path()*, checks if it exists already, creates it and installs requirements otherwise. The virtualenvs are stored in a folder based on the *Arca* `base_dir` setting.

Parameters **path** – `Path` to the cloned repository.

get_virtualenv_path(*requirements_option: arca.backend.base.RequirementsOptions*, *requirements_hash: Optional[str]*) → pathlib.Path
Returns the path to the virtualenv the current state of the repository.

8.4.4 Docker

class arca.**DockerBackend**(***kwargs*)
Bases: `arca.backend.base.BaseBackend`

Runs the tasks in Docker containers.

Available settings:

- **python_version** - set a specific version, current env. python version by default
- **keep_container_running** - stop the container right away (default) or keep it running
- **apt_dependencies** - a list of dependencies to install via `apt-get`
- **disable_pull** - build all locally
- **inherit_image** - instead of using the default base Arca image, use this one
- **use_registry_name** - use this registry to store images with requirements and dependencies
- **registry_pull_only** - only use the registry to pull images, don't push updated

build_image(*image_name: str*, *image_tag: str*, *repo_path: pathlib.Path*, *requirements_option: arca.backend.base.RequirementsOptions*, *dependencies: Optional[List[str]]*)
Builds an image for specific requirements and dependencies, based on the settings.

Parameters

- **image_name** – How the image should be named
- **image_tag** – And what tag it should have.
- **repo_path** – Path to the cloned repository.
- **requirements_option** – How requirements are set in the repository.
- **dependencies** – List of dependencies (in the formalized format)

Returns The Image instance.

Return type docker.models.images.Image

build_image_from_inherited_image(*image_name: str*, *image_tag: str*, *repo_path: pathlib.Path*, *requirements_option: arca.backend.base.RequirementsOptions*)
Builds a image with installed requirements from the inherited image. (Or just tags the image if there are no requirements.)

See *build_image()* for parameters descriptions.

Return type docker.models.images.Image

check_docker_access()
Creates a `DockerClient` for the instance and checks the connection.

Raises ***BuildError*** – If docker isn't accessible by the current user.

container_running(*container_name*)
Finds out if a container with name `container_name` is running.

Returns `Container` if it's running, `None` otherwise.

Return type Optional[docker.models.container.Container]

get_arca_base(*pull=True*)
Returns the name and tag of image that has the basic build dependencies installed with just pyenv installed, with no python installed. (Builds or pulls the image if it doesn't exist locally.)

get_container_name(*repo: str*, *branch: str*, *git_repo: git.repo.base.Repo*)
Returns the name of the container used for the repo.

get_dependencies() → Optional[List[str]]
Returns the `apt_dependencies` setting to a standardized format.

Raises ***ArcaMisconfigured*** – if the dependencies can't be converted into a list of strings

Returns List of dependencies, `None` if there are none.

get_dependencies_hash(*dependencies*)
Returns a SHA1 hash of the dependencies for usage in image names/tags.

get_image(*image_name*, *image_tag*)
Returns a `Image` instance for the provided name and tag.

Return type docker.models.images.Image

get_image_for_repo(*repo: str*, *branch: str*, *git_repo: git.repo.base.Repo*, *repo_path: pathlib.Path*)
Returns an image for the specific repo (based on settings and requirements).

1. Checks if the image already exists locally
2. Tries to pull it from registry (if `use_registry_name` is set)
3. Builds the image
4. Pushes the image to registry so the image is available next time (if `registry_pull_only` is not set)

See *`run()`* for parameters descriptions.

Return type docker.models.images.Image

get_image_name(*repo_path: pathlib.Path*, *requirements_option: arca.backend.base.RequirementsOptions*, *dependencies: Optional[List[str]]*) → str
Returns the name for images with installed requirements and dependencies.

get_image_tag(*requirements_option: arca.backend.base.RequirementsOptions*, *requirements_hash: Optional[str]*, *dependencies: Optional[List[str]]*) → str
Returns the tag for images with the dependencies and requirements installed.

64-byte hexadecimal strings cannot be used as docker tags, so the prefixes are necessary. Double hashing the dependencies and requirements hash to make the final tag shorter.

Prefixes:

- Image type:

  - i – Inherited image
  - a – Arca base image
- Requirements:
  - r – Does have some kind of requirements
  - s – Doesn't have requirements
- Dependencies:
  - d – Does have dependencies
  - e – Doesn't have dependencies

Possible outputs:

- Inherited images:
  - *ise* – no requirements
  - *ide_<hash(requirements)>* – with requirements
- From Arca base image:
  - *<Arca version>_<Python version>_ase* – no requirements and no dependencies
  - *<Arca version>_<Python version>_asd_<hash(dependencies)>* – only dependencies
  - *<Arca version>_<Python version>_are_<hash(requirements)>* – only requirements
  - *<Arca version>_<Python version>_ard_<hash(hash(dependencies) + hash(requirements))>* – both requirements and dependencies

get_image_with_installed_dependencies(*image_name: str, dependencies: Optional[List[str]]*) → Tuple[str, str]

Return name and tag of a image, based on the Arca python image, with installed dependencies defined by `apt_dependencies`.

Parameters

- **image_name** – Name of the image which will be ultimately used for the image.
- **dependencies** – List of dependencies in the standardized format.

get_inherit_image() → Tuple[str, str]

Parses the `inherit_image` setting, checks if the image is present locally and pulls it otherwise.

Returns Returns the name and the tag of the image.

Raises **ArcaMisconfiguration** – If the image can't be pulled from registries.

get_install_requirements_dockerfile(*name: str, tag: str, repo_path: pathlib.Path, requirements_option: arca.backend.base.RequirementsOptions*) → str

Returns the content of a Dockerfile that will install requirements based on the repository, prioritizing Pipfile or Pipfile.lock and falling back on requirements.txt files

get_or_build_image(*name: str, tag: str, dockerfile: Union[str, Callable[..., str]], *, pull=True, build_context: Optional[pathlib.Path] = None*)

A proxy for commonly built images, returns them from the local system if they exist, tries to pull them if pull isn't disabled, otherwise builds them by the definition in `dockerfile`.

Parameters

- **name** – Name of the image

- **tag** – Image tag
- **dockerfile** – Dockerfile text or a callable (no arguments) that produces Dockerfile text
- **pull** – If the image is not present locally, allow pulling from registry (default is `True`)
- **build_context** – A path to a folder. If it's provided, docker will build the image in the context of this folder. (eg. if `ADD` is needed)

get_python_base(*python_version*, *pull=True*)
Returns the name and tag of an image with specified `python_version` installed, if the image doesn't exist locally, it's pulled or built (extending the image from *get_arca_base()*).

get_python_version() → str
Returns either the specified version from settings or platform.python_version()

image_exists(*image_name*, *image_tag*)
Returns if the image exists locally.

push_to_registry(*image*, *image_tag: str*)
Pushes a local image to a registry based on the `use_registry_name` setting.

Raises ***PushToRegistryError*** – If the push fails.

run(*repo: str*, *branch: str*, *task: arca.task.Task*, *git_repo: git.repo.base.Repo*, *repo_path: pathlib.Path*) → arca.result.Result
Gets or builds an image for the repo, gets or starts a container for the image and runs the script.

Parameters

- **repo** – Repository URL
- **branch** – Branch ane
- **task** – *Task* to run.
- **git_repo** – `Repo` of the cloned repository.
- **repo_path** – `Path` to the cloned location.

start_container(*image*, *container_name: str*, *repo_path: pathlib.Path*)
Starts a container with the image and name `container_name` and copies the repository into the container.

Return type docker.models.container.Container

stop_containers()
Stops all containers used by this instance of the backend.

tar_files(*path: pathlib.Path*) → bytes
Returns a tar with the git repository.

tar_runner()
Returns a tar with the runner script.

tar_task_definition(*name: str*, *contents: str*) → bytes
Returns a tar with the task definition.

Parameters

- **name** – Name of the file
- **contents** – Contens of the definition, utf-8

try_pull_image_from_registry(*image_name*, *image_tag*)
Tries to pull a image with the tag `image_tag` from registry set by `use_registry_name`. After the image is pulled, it's tagged with `image_name:image_tag` so lookup can be made locally next time.

Returns A `Image` instance if the image exists, `None` otherwise.

Return type Optional[docker.models.images.Image]

validate_configuration()
Validates the provided settings.

- Checks `inherit_image` format.
- Checks `use_registry_name` format.
- Checks that `apt_dependencies` is not set when `inherit_image` is set.

Raises ***ArcaMisconfigured*** – If some of the settings aren't valid.

8.4.5 Vagrant

class arca.**VagrantBackend**(***kwargs*)
Bases: `arca.backend.docker.DockerBackend`

Uses Docker in Vagrant.

Inherits settings from *DockerBackend*:

- **python_version**
- **apt_dependencies**
- **disable_pull**
- **inherit_image**
- **use_registry_name**
- **keep_containers_running** - applies for containers inside the VM, default being `True` for this backend

Adds new settings:

- **box** - what Vagrant box to use (must include docker >= 1.8 or no docker), `ailispaw/barge` being the default
- **provider** - what provider should Vagrant user, `virtualbox` being the default
- **quiet** - Keeps the extra vagrant logs quiet, `True` being the default
- **keep_vm_running** - Keeps the VM up until *stop_vm()* is called, `False` being the default
- **destroy** - Destroy the VM (instead of halt) when stopping it, `False` being the default

ensure_vm_running(*vm_location*)
Gets or creates a Vagrantfile in `vm_location` and calls `vagrant up` if the VM is not running.

fabric_task
Returns a fabric task which executes the script in the Vagrant VM

get_vm_location() → pathlib.Path
Returns a directory where a Vagrantfile should be - folder called `vagrant` in the Arca base dir.

init_vagrant(*vagrant_file*)
Creates a Vagrantfile in the target dir, with only the base image pulled. Copies the runner script to the directory so it's accessible from the VM.

inject_arca(*arca*)
Apart from the usual validation stuff it also creates log file for this instance.

run(*repo: str*, *branch: str*, *task: arca.task.Task*, *git_repo: git.repo.base.Repo*, *repo_path: pathlib.Path*)
Starts up a VM, builds an docker image and gets it to the VM, runs the script over SSH, returns result. Stops the VM if `keep_vm_running` is not set.

stop_containers()
Raises an exception in this backend, can't be used. Stop the entire VM instead.

stop_vm()
Stops or destroys the VM used to launch tasks.

validate_configuration()
Runs *arca.DockerBackend.validate_configuration()* and checks extra:

- `box` format
- `provider` format
- `use_registry_name` is set and `registry_pull_only` is not enabled.

8.5 Exceptions

exception arca.exceptions.**ArcaException**
Bases: `Exception`

A base exception from which all exceptions raised by Arca are subclassed.

exception arca.exceptions.**ArcaMisconfigured**
Bases: `ValueError`, *arca.exceptions.ArcaException*

An exception for all cases of misconfiguration.

exception arca.exceptions.**BuildError**(**args*, *extra_info=None*, ***kwargs*)
Bases: *arca.exceptions.ArcaException*

Raised if the task fails.

extra_info = None
Extra information what failed

exception arca.exceptions.**BuildTimeoutError**(**args*, *extra_info=None*, ***kwargs*)
Bases: *arca.exceptions.BuildError*

Raised if the task takes too long.

exception arca.exceptions.**FileOutOfRangeError**
Bases: `ValueError`, *arca.exceptions.ArcaException*

Raised if `relative_path` in *Arca.static_filename* leads outside the repository.

exception arca.exceptions.**PullError**
Bases: *arca.exceptions.ArcaException*

Raised if repository can't be cloned or pulled.

exception arca.exceptions.**PushToRegistryError**(**args*, *full_output=None*, ***kwargs*)
Bases: *arca.exceptions.ArcaException*

Raised if pushing of images to Docker registry in `DockerBackend` fails.

full_output = None
Full output of the push command

exception arca.exceptions.**RequirementsMismatch**(**args*, *diff=None*, ***kwargs*)
Bases: ValueError, *arca.exceptions.ArcaException*

Raised if the target repository has extra requirements compared to the current environment if the `requirements_strategy` of `CurrentEnvironmentBackend` is set to `arca.backends.RequirementsStrategy.raise`.

diff = None
The extra requirements

exception arca.exceptions.**TaskMisconfigured**
Bases: ValueError, *arca.exceptions.ArcaException*

Raised if Task is incorrectly defined.

8.6 Utils

class arca.utils.**LazySettingProperty**(*, *key=None*, *default=NOT_SET*, *convert: Callable = None*)
For defining properties for the `Arca` class and for the backends. The property is evaluated lazily when accessed, getting the value from settings using the instances method `get_setting`. The property can be overridden by the constructor.

exception SettingsNotReady

class arca.utils.**NotSet**
For default values which can't be `None`.

class arca.utils.**Settings**(*data: Optional[Dict[str, Any]] = None*)
A class for handling *Arca* settings.

get(**keys*, *default: Any = NOT_SET*) → Any
Returns values from the settings in the order of keys, the first value encountered is used.

Example:

```
>>> settings = Settings({"ARCA_ONE": 1, "ARCA_TWO": 2})
>>> settings.get("one")
1
>>> settings.get("one", "two")
1
>>> settings.get("two", "one")
2
>>> settings.get("three", "one")
1
>>> settings.get("three", default=3)
3
>>> settings.get("three")
Traceback (most recent call last):
...
KeyError:
```

Parameters

- **keys** – One or more keys to get from settings. If multiple keys are provided, the value of the first key that has a value is returned.
- **default** – If none of the `options` aren't set, return this value.

Returns A value from the settings or the default.

Raises

- **ValueError** – If no keys are provided.
- **KeyError** – If none of the keys are set and no default is provided.

arca.utils.**get_hash_for_file**(*repo: git.repo.base.Repo, path: Union[str, pathlib.Path]*) → str
Returns the hash for the specified path.

Equivalent to `git rev-parse HEAD:X`

Parameters

- **repo** – The repo to check in
- **path** – The path to a file or folder to get hash for

Returns The hash

arca.utils.**get_last_commit_modifying_files**(*repo: git.repo.base.Repo*, **files*) → str
Returns the hash of the last commit which modified some of the files (or files in those folders).

Parameters

- **repo** – The repo to check in.
- **files** – List of files to check

Returns Commit hash.

arca.utils.**is_dirty**(*repo: git.repo.base.Repo*) → bool
Returns if the `repo` has been modified (including untracked files).

arca.utils.**load_class**(*location: str*) → type
Loads a class from a string and returns it.

```
>>> from arca.utils import load_class
>>> load_class("arca.backend.BaseBackend")
<class 'arca.backend.base.BaseBackend'>
```

Raises ***ArcaMisconfigured*** – If the class can't be loaded.

CHAPTER 9

Changes

9.1 0.3.3 (2019-12-10)

Changes:

- Updated dependencies
- Allowed branches with slashes (#79)

9.2 0.3.2 (2019-11-23)

Changes:

- Moved the project under organisation Pyvec.
- Changed the Docker registry for the base images to *arcaoss/arca*.
- Fixed unicode paths to repositories (#60)

9.3 0.3.1 (2018-11-16)

Raising a Arca exception when building of a Docker image fails. (#56, #57)

9.4 0.3.0 (2018-08-25)

Changes:

- Removed CurrentEnvironmentBackend's capability to process requirements - all requirements are ignored. (**BACKWARDS INCOMPATIBLE**)

- Added support for installing requirements using Pipenv. The directory containing `Pipfile` and `Pipfile.lock` is set by the backend option **pipfile_location**, by default the root of the repository is selected. The Pipenv files take precedence over regular requirement files.
- The `Result` class now has two more attributes, `stdout` and `stderr` with the outputs of launched tasks to standard output and error. Priting is therefore now allowed in the endpoints.
- Using UTF-8 locale in Docker images used in `DockerBackend`.
- Supporting Python 3.7.

9.5 0.2.1 (2018-06-11)

Updated dogpile.cache to 0.6.5, enabling compatability with Python 3.7. Updated the Docker backend to be able to run on Python betas.

9.6 0.2.0 (2018-05-09)

This release has multiple backwards incompatible changes against the previous release.

Changes:

- Using extras to install Docker and Vagrant dependencies
  - `pip install arca[docker]` or `pip install arca[vagrant]` has to be used
- Automatically using cloned repositories as reference for newly cloned branches
- Using Debian as the default base image in the Docker backend:
  - **apk_dependencies** changed to **apt_dependencies**, now installing using *apt-get*
- Vagrant backend only creates one VM, instead of multiple – see its documentation
- Added timeout to tasks, 5 seconds by default. Can be set using the argument **timeout** for `Task`.
- Added timeout to installing requirements, 300 seconds by default. Can be set using the **requirements_timeout** configuration option for backends.

9.7 0.1.1 (2018-04-23)

Updated gitpython to 2.1.9

9.8 0.1.0 (2018-04-18)

Initial release

Changes:

- Updated PyPI description and metadata

9.9 0.1.0a0 (2018-04-13)

Initial alfa release

Python Module Index

a

Index

H

I

L

M

N

O

P

R

S

T

V