# Runway Model SDK

***Release v0.1.0***

**Runway AI, Inc.**

**Mar 28, 2022**

# CONTENTS

These documents serve as a reference for the Runway Model SDK. With a few lines of code, you can port existing ML models to the Runway platform so they can be used and shared by others.

CHAPTER

ONE

# INSTALLING

This SDK supports both Python 3.6+. You can install the module using either `pip` or `pip3` like so:

```
pip3 install runway-python
```

Published versions of the SDK are hosted on the PyPI project website.

CHAPTER

TWO

# RUNWAY MODELS

A Runway model consists of two special files:

- `runway_model.py`: A python script that imports the `runway` module (SDK) and exposes its interface via one or more `runway.command()` functions. This file is used as the **entrypoint** to your model.
- `runway.yml`: A configuration file that describes dependencies and build steps needed to build and run the model.

## 2.1 Example `runway_model.py`

Runway models expose a standard interface that allows the Runway app to interact with them over HTTP. This is accomplished using three functions: `@runway.setup()`, `@runway.command()`, and `runway.run()`.

Any Python-based model, independent of the ML framework or toolkit, can be converted into a Runway model using this simple interface. For more information about the `runway` module, see the module reference page.

**Note:** This is example code for demonstration purposes only. It will not run, as the `your_image_generation_model` import is not a real python module.

```
import runway
from runway.data_types import category, vector, image
from your_image_generation_model import big_model, little_model

# The setup() function runs once when the model is initialized, and will run
# again for each well formed HTTP POST request to http://localhost:9000/setup.
@runway.setup(options={'model_size': category(choices=['big', 'little'])})
def setup(opts):
    if opts['model_size'] == 'big':
        return big_model()
    else:
        return little_model()

inputs = { 'noise_vector': vector(length=128, description='A random seed.') }
outputs = { 'image': image(width=512, height=512) }

# The @runway.command() decorator is used to create interfaces to call functions
# remotely via an HTTP endpoint. This lets you send data to, or get data from,
# your model. Each command creates an HTTP route that the Runway app will use
# to communicate with your model (e.g. POST /generate). Multiple commands
# can be defined for the same model.
@runway.command('generate', inputs=inputs, outputs=outputs, description='Generate an␣
↪image.')
```

(continues on next page)

(continued from previous page)

```
def generate(model, input_args):
    # Functions wrapped by @runway.command() receive two arguments:
    # 1. Whatever is returned by a function wrapped by @runway.setup(),
    #    usually a model.
    # 2. The input arguments sent by the remote caller via HTTP. These values
    #    match the schema defined by inputs.
    img = input_args['image']
    return model.generate(img)

# The runway.run() function triggers a call to the function wrapped by
# @runway.setup() passing model_options as its single argument. It also
# creates an HTTP server that listens for and fulfills remote requests that
# trigger commands.
if __name__ == '__main__':
    runway.run(host='0.0.0.0', port=9000, model_options={ 'model_size': 'big' })
```

If you are looking to port your own model, we recommend starting from our Model Template repository hosted on GitHub. This repository contains a basic model that you can use as boilerplate instead of having to start from scratch.

## 2.2 Example `runway.yml`

Each Runway model must have a `runway.yml` configuration file in its root directory. This file defines the steps needed to build and run your model for use with the Runway app. This file is written in YAML, a human-readable superset of JSON. Below is an example `runway.yml` file. This example file illustrates how you can provision your model's environment.

```
version: 0.1
python: 3.6
entrypoint: python runway_model.py
cuda: 9.2
framework: tensorflow
files:
    ignore:
        - image_dataset/*
build_steps:
    - pip install runway-python
    - pip install -r requirements.txt
```

Continue on to the Runway YAML reference page to learn more about the possible configuration values supported by the `runway.yml` file, or hop over to the Example Models page to check out the source code for some of the models that have already been ported to Runway.

### 2.2.1 Runway YAML File

A `runway.yml` file is required to be in the root file directory of each Runway model. This file provides instructions for defining an environment, installing dependencies, and running your model in a standard and reproducible manner. These instructions are used by the Runway build pipeline to create a Docker image that can be run in a platform independent way on a local machine or a remote GPU cloud instance (although this process is abstracted away from the model builder). The `runway.yml` config file is written in YAML and has a simple structure. You are free to copy and paste the example config file below, changing or removing the values that you need.

---

**Note:** The Runway configuration file must be named `runway.yml` and exist in the root (top-level) directory of your

model. Also, make sure you use the `.yml` file extension as the alternative `.yaml` extension is not supported.

### Example

```yaml
# Specify the version of the runway.yml spec.
version: 0.1
# Supported python versions are: 3.6
python: 3.6
# The command to run your model. This value is used as the CMD value in
# the generated Docker image.
entrypoint: python runway_model.py
# Which NVIDIA CUDA version to use. Supported versions include 10.2, 10, 9.2, and 9.
cuda: 9.2
# Which ML framework would you like to pre-install? The appropriate GPU/CPU
# versions of these libraries are selected automatically. Accepts values
# "tensorflow" and "pytorch", installinv Tensorflow v1.12 and Pytorch v1.0
# respectively.
framework: tensorflow
# Builds are created for CPU and GPU environments by default. You can use the
# spec object to limit your builds to one environment if you'd like, for
# instance if your model doesn't use CUDA or run on a GPU you can set
# gpu: False.
spec:
    cpu: True
    gpu: True
files:
    # All files in the root project directory will be copied to the Docker image
    # automatically. Builds that require excessive storage can fail or take a
    # very long time to install on another user's machine. You can use the
    # files.ignore array to exclude files from your build.
    ignore:
        - my_dataset/*
        - secrets.txt
# The build_steps array allows you to run shell commands at build time. Each
# Each build step is executed in the order it appears in the array.
build_steps:
    # We recommend pinning to a specific version of the Runway Model SDK until
    # the first major release, as breaking changes may be introduced to the SDK
    - pip install runway-python==0.0.74
    - pip install -r requirements.txt
    # The if_gpu and if_cpu directives can be used to run build steps
    # conditionally depending on the build environment.
    - if_gpu: echo "Building in a GPU environment..."
    - if_cpu: echo "Building in a CPU only environment..."
```

**Note:** If you require an ML framework other than Tensorflow or Pytorch, or a version of these libraries that is different than the versions provided by the `frameworks` object, you can install these dependencies manually in the build steps.

```yaml
build_steps:
    - pip install tensorflow==1.0
    - if_gpu: pip install tensorflow-gpu==1.0
```

#### Schema Reference

- `version` (int, optional, default = `0.1`): This version specifies the schema of the configuration file not the version of the Runway Model SDK itself.
- `python` (float, **required**): The Python version to use when running the model installing python dependencies. Only `3.6` is supported at this time.
- `entrypoint` (string, **required**): The command to run your model. This value is used as the CMD value in the generated Docker image. A standard value for this field might be `entrypoint: python runway_model.py` where `runway_model.py` implements the `@runway.setup()`, `@runway.command()`, and most importantly the `runway.run()` functions.
- `cuda` (float, **required if building for GPU**): The NVIDIA CUDA version to use in the production GPU runtime environment. The currently supported CUDA versions are `10.2`, `10`, `9.2`, and `9`.
- `framework` (string, optional, default = `None`): The machine learning framework to pre-install during the build. Currently we support `"tensorflow"` and `"pytorch"` which will install the appropriate CPU or GPU packages of Tensorflow v1.12.0 and Pytorch v1.0 respectively depending on the build environment. If you require an ML framework other than Tensorflow or Pytorch, or a version of these libraries that is different than the versions provided by the `frameworks` object, you can omit this object and install these dependencies manually in the build steps.
- `spec` (object, optional): A dictionary of boolean values specifying which CPU/GPU environments to build for. Both the `cpu` and `gpu` environments are enabled (`True`) by default.
  - `cpu` (boolean, optional, default = `True`): Create a CPU build.
  - `gpu` (boolean, optional, default = `True`): Create a GPU build.
- `files` (object, optional): A dictionary that defines special behaviors for certain files. All values in this dictionary are specified as paths, with support for the glob character (e.g. `data/*.jpg`).
  - `ignore` (array of strings, optional): A list of file paths to exclude from the build.
- `build_steps` (array of strings or dictionary values containing the `if_cpu` and `if_gpu` keys, optional): A list of shell commands to run at build time. Use this list to define custom build steps. Build steps are run in the order they appear in the array. The `if_gpu` and `if_cpu` directives can be used to run build steps conditionally depending on the build environment.

### 2.2.2 Runway Module

The Runway module exposes three simple functions that can be combined to expose your models to the Runway app using a simple interface.

- *`@runway.setup()`*: A Python decorator used to initialize and configure your model.
- *`@runway.command()`*: A Python decorator used to define the interface to your model. Each command creates an HTTP route which can process user input and return outputs from the model.
- *`runway.run()`*: The entrypoint function that starts the SDK's HTTP interface. It fires the function decorated by `@runway.setup()` and listens for commands on the network, forwarding them along to the appropriate functions decorated with `@runway.command()`.

#### Reference

`runway.`**`setup`**(*decorated_fn=None*, *options=None*)

This decorator is used to wrap your own `setup()` (or equivalent) function to run whatever initialization code

you'd like. Your wrapped function *should* configure and return a model. Your function *should* also be made resilient to being called multiple times, as this is possible depending on the behavior of the client application.

This endpoint exposes a `/setup` HTTP route and calls the wrapped (decorated) function once on `runway.run()` and whenever a new POST request is made to the `/setup` route.

```
import runway
from runway.data_types import category
from your_code import model

# Descriptions are used to document each data type; Their values appear in the␣
↪app as
# tooltips. Writing detailed descriptions for each model option goes a long way␣
↪towards
# helping users learn how to interact with your model. Write descriptions as full␣
↪sentences.
network_size_description = "The size of the network. A larger number will result␣
↪in better accuracy at the expense of increased latency."
options = { "network_size": category(choices=[64, 128, 256, 512], default=256,␣
↪description=network_size_description) }
@runway.setup(options=options)
def setup(opts):
    print("Setup ran, and the network size is {}".format(opts["network_size"]))
    return model(network_size=opts["network_size"])
```

**Note:** This is example code for demonstration purposes only. It will not run, as the `your_code` import is not a real python module.

**Parameters**

- **decorated_fn** (*function, optional*) – A function to be decorated. This argument is automatically assigned the value of the wrapped function if the decorator syntax is used without a function call (e.g. `@runway.setup` instead of `@runway.setup()`).
- **options** (*dict, optional*) – A dictionary of setup options, mapping string names to `runway.data_types`. These options define the schema of the object that is passed as the single argument to the wrapped function.

**Returns** A decorated function

**Return type** function or *NoneType*

`runway.`**command**(*name*, *inputs={}*, *outputs={}*)

This decorator function is used to define the interface for your model. All functions that are wrapped by this decorator become exposed via HTTP requests to `/<command_name>`. Each command that you define can be used to get data into or out of your runway model, or trigger an action.

```
import runway
from runway.data_types import category, vector, image
from your_code import model

@runway.setup
def setup():
    return model()

sample_inputs= {
```

(continues on next page)

(continued from previous page)

```
    "z": vector(length=512, description="The seed used to generate an output␣
↪image."),
    "category": category(choices=["day", "night"])
}

sample_outputs = {
    "image": image(width=1024, height=1024)
}

# Descriptions are used to document each data type and `@runway.command()`; Their␣
↪values appear
# in the app as tooltips. Writing detailed descriptions for each model option␣
↪goes a long way
# towards helping users learn how to interact with your model. Write descriptions␣
↪as full
# sentences.
sample_description = "Generate a new image based on a z-vector and an output␣
↪style: Day or night."
@runway.command("sample",
                inputs=sample_inputs,
                outputs=sample_outputs,
                description=sample_description)
def sample(model, inputs):
    # The parameters passed to a function decorated by @runway.command() are:
    #   1. The return value of a function wrapped by @runway.setup(), usually a␣
↪model.
    #   2. The inputs sent with the HTTP request to the /<command_name> endpoint,
    #      as defined by the inputs keyword argument delivered to @runway.
↪command().
    img = model.sample(z=inputs["z"], category=inputs["category"])
    # `img` can be a PIL or numpy image. It will be encoded as a base64 URI string
    # automatically by @runway.command().
    return { "image": img }
```

**Note:** All `@runway.command()` decorators accept a `description` keyword argument that can be used to describe what the command does. Descriptions appear as tooltips in the app and help users learn what the command does. It's best practice to write full-sentence descriptions for each of your input variables and commands.

**Parameters**

- **name** (*string*) – The name of the command. This name is used to create the HTTP route associated with the command (i.e. a name of "generate_text" will generate a `/generate_text` route).
- **inputs** (*dict*) – A dictionary mapping input names to `runway.data_types`. This dictionary defines the interface used to send input data to this command. At least one key value pair is required.
- **outputs** (*dict*) – A dictionary defining the output data returned from the wrapped function as `runway.data_types`. At least one key value pair is required.
- **description** (*string, optional*) – A text description of what this command does. If this parameter is present its value will be rendered as a tooltip in Runway. Defaults to None.

**Raises** **Exception** – An exception if there isn't at least one key value pair for both inputs and outputs dictionaries

**Returns** A decorated function

**Return type** function

runway.**run**(*host='0.0.0.0'*, *port=9000*, *model_options={}*, *debug=False*, *meta=False*)

Run the model and start listening for HTTP requests on the network. By default, the server will run on port `9000` and listen on all network interfaces (`0.0.0.0`).

```
import runway

# ... setup an initialization function with @runway.setup()
# ... define a command or two with @runway.command()

# now it's time to run the model server which actually sets up the
# routes and handles the HTTP requests.
if __name__ == "__main__":
    runway.run()
```

`runway.run()` acts as the entrypoint to the runway module. You should call it as the last thing in your `runway_model.py`, once you've defined a `@runway.setup()` function and one or more `@runway.command()` functions.

**Parameters**

- **host** (*string, optional*) – The IP address to bind the HTTP server to, defaults to `"0.0.0.0"` (all interfaces). This value will be overwritten by the `RW_HOST` environment variable if it is present.
- **port** (*int, optional*) – The port to bind the HTTP server to, defaults to `9000`. This value will be overwritten by the `RW_PORT` environment variable if it is present.
- **model_options** (*dict, optional*) – The model options that are passed to `@runway.setup()` during initialization, defaults to `{}`. This value will be overwritten by the `RW_MODEL_OPTIONS` environment variable if it is present.
- **debug** (`boolean`, *optional*) – Whether to run the Flask HTTP server in debug mode, which enables live reloading, defaults to `False`. This value will be overwritten by the `RW_DEBUG` environment variable if it is present.
- **meta** (`boolean`, *optional*) – Print the model's options and commands as JSON and exit, defaults to `False`. This functionality is used in a production environment to dynamically discover the interface presented by the Runway model at runtime. This value will be overwritten by the `RW_META` environment variable if it is present.
- **no_serve** – Don't start the Flask server, defaults to `False` (i.e. the Flask server is started by default when the `runway.run()` function is called without setting this argument set to True). This functionality is used during automated testing to mock HTTP requests using Flask's `app.test_client()` (see Flask's testing docs for more details).

**Warning:** All keyword arguments to the `runway.run()` function will be overwritten by environment variables when your model is run by the Runway app. Using the default values for these arguments, or supplying your own in python code, is fine so long as you are aware of the fact that their values will be overwritten by the following environment variables at runtime in a production environment:

- `RW_HOST`: Defines the IP address to bind the HTTP server to. This environment variable overwrites any value passed as the `host` keyword argument.

- `RW_PORT`: Defines the port to bind the HTTP server to. This environment variable overwrites any value passed as the `port` keyword argument.
- `RW_MODEL_OPTIONS`: Defines the model options that are passed to `@runway.setup()` during initialization. This environment variable overwrites any value passed as the `model_options` keyword argument.
- `RW_DEBUG`: Defines whether to run the Flask HTTP server in debug mode, which enables live reloading. This environment variable overwrites any value passed as the `debug` keyword argument. `RW_DEBUG=1` enables debug mode.
- `RW_META`: Defines the behavior of the `runway.run()` function. If `RW_META=1` the function prints the model's options and commands as JSON and then exits. This environment variable overwrites any value passed as the `meta` keyword argument.
- `RW_NO_SERVE`: Forces `runway.run()` to not start its Flask server. This environment variable overwrites any value passed as the `no_serve` keyword argument.

### 2.2.3 Data Types

The Runway Model SDK provides several data types that can be used to pass values to and from runway models and the applications that control them. The data types currently supported by the SDK are `number`, `text`, `image`, `array`, `vector`, `category`, `file`, and `any`, an extensible data type. These data types are primarily used in two places:

- The `options` parameter in the `@runway.setup()` decorator
- The `input` and `output` parameters in `@runway.command()` decorator

---

**Note:** This is example code for demonstration purposes only. It will not run, as the `your_code` import is not a real python module.

---

```
import runway
from runway.data_types import category, vector, image
from your_code import model

options = {"network_size": category(choices=[64, 128, 256, 512], default=256)}
@runway.setup(options=options)
def setup(opts):
    return model(network_size=opts["network_size"])


sample_inputs= {
    "z": vector(length=512),
    "category": category(choices=["day", "night"])
}

sample_outputs = {
    "image": image(width=1024, height=1024)
}

@runway.command("sample", inputs=sample_inputs, outputs=sample_outputs)
def sample(model, inputs):
    img = model.sample(z=inputs["z"], category=inputs["category"])
```

(continues on next page)

(continued from previous page)

```
    # `img` can be a PIL or numpy image. It will be encoded as a base64 URI
    # string automatically by @runway.command().
    return { "image": img }

if __name__ == "__main__":
    runway.run()
```

## Reference

**class** runway.data_types.**BaseType**(*data_type*, *description=None*)

An abstract class that defines a base data type interface. This type should be used as the base class of new data types, never directly.

**Parameters**

- **data_type** (*string*) – The data type represented as a string
- **description** (*string, optional*) – A description of this variable and how its used in the model, defaults to None

**class** runway.data_types.**any**(*description=None*)

A generic data type. The value this data type takes must be serializable to JSON.

```
import yaml
import runway
from runway.data_types import any
from your_code import model

# an example of passing your own yaml configuration using an "any" data_type and␣
↪PyYAML
@runway.setup(options={ "configuration": any() })
def setup(opts):
    # parse the configuration string as yaml, and then pass the resulting
    # object as the configuration to your model
    config = yaml.load(opts["configuration"])
    return model(config)
```

**Parameters** **description** (*string, optional*) – A description of this variable and how its used in the model, defaults to None

**class** runway.data_types.**array**(*item_type=None*, *description=None*, *min_length=0*, *max_length=None*)

A data type representing an array (list) of other runway.data_type objects.

```
import runway
from runway.data_types import array, text

@runway.setup(options={ "seed_sentences": array(item_type=text, min_length=5) })
def setup(opts):
    for i in range(5):
        print("Sentence {} is "{}"".format(i+1, opts["seed_sentences"][i]))
```

**Parameters**

- **item_type** (*runway.data_type*) – A runway.data_type class, or an instance of a runway.data_type class

- **description** (*string, optional*) – A description of this variable and how its used in the model, defaults to None
- **min_length** (*int, optional*) – The minimum number of elements required to be in the array, defaults to 0
- **max_length** (*int, optional*) – The maximum number of elements allowed to be in the array, defaults to None

**Raises** ***MissingArgumentError*** – A missing argument error if item_type is not specified

**class** runway.data_types.**boolean**(*description=None*, *default=False*)

A basic boolean data type. The only accepted values for this data type are *True* and *False*.

```
import runway
from runway.data_types import boolean

@runway.setup(options={ "crop": boolean(default=True) })
def setup(opts):
    if opts["crop"]:
        print("The user has chosen to crop the image.")
    else:
        print("The user has chosen not to crop the image.")
```

**Parameters**

- **description** (*string, optional*) – A description of this variable and how it's used in the model, defaults to None
- **default** (*bool, optional*) – A default value for this boolean variable, defaults to False

**class** runway.data_types.**category**(*description=None*, *choices=None*, *default=None*)

A categorical data type that allows you to specify a variable's value as a member of a set list of choices.

```
import runway
from runway.data_types import category

# if no default value is specified, the first element in the choices
# list will be used
cat = category(choices=["rgb", "bgr", "rgba", "bgra"], default="rgba")
@runway.setup(options={ "pixel_order": cat })
def setup(opts):
    print("The selected pixel order is {}".format(opts["pixel_order"]))
```

**Parameters**

- **description** (*string, optional*) – A description of this variable and how its used in the model, defaults to None
- **choices** (*A list of strings*) – A list of categories, defaults to None
- **default** (*A list of strings*) – A default list of categories, defaults to None

**Raises**

- ***MissingArgumentError*** – A missing argument error if choices is not a list with at least one element.

- ***InvalidArgumentError*** – An invalid argument error if a default argument is specified and that argument does not appear in the choices list.

**class** runway.data_types.**directory**(*description=None*, *default=None*)

A data type that represents a file directory. It can be a local path on disk or a remote tarball loaded over HTTP. .. code-block:: python

import runway from runway.data_types import directory

@runway.setup(options={"checkpoint_dir": directory}) def setup(opts):

model = initialize_model_from_checkpoint_folder(args["checkpoint_dir"]) return model

**Parameters** **description** (*string, optional*) – A description of this variable and how its used in the model, defaults to None

**class** runway.data_types.**file**(*description=None*, *is_directory=False*, *extension=None*, *default=None*)

A data type that represents a file or directory. The file can be a local resource on disk or a remote resource loaded over HTTP. Instantiate this class to create a new runway model variable.

```
import runway
from runway.data_types import file

@runway.setup(options={"checkpoint": file(extension=".h5")})
def setup(opts):
    model = initialize_model_from_checkpoint(args["checkpoint"])
    return model
```

**Parameters**

- **description** (*string, optional*) – A description of this variable and how its used in the model, defaults to None
- **is_directory** (*bool, optional*) – Does this variable represent a directory instead of a file? Defaults to False.
- **extension** (*string, optional*) – Accept only files of this extension.
- **default** (*string, optional*) – Use this path if no input file or directory is provided.

**class** runway.data_types.**image**(*description=None*, *channels=3*, *min_width=None*, *min_height=None*, *max_width=None*, *max_height=None*, *width=None*, *height=None*, *default_output_format=None*)

A data type representing an image. Images represent PIL or numpy images but are passed to and from the Model SDK as base64 encoded data URI strings over the network (e.g. `...`).

When using an image as an output data type for a function wrapped by `@runway.command()`, return a PIL or numpy image from your wrapped function and it will automatically be serialized as a base64 encoded data URI.

```
import runway
from runway.data_types import image

inputs = {"image": image(width=512, height=512)}
outputs = {"image": image(width=512, height=512)}
@runway.command("style_transfer", inputs=inputs, outputs=outputs)
```

(continues on next page)

(continued from previous page)

```
def style_transfer(result_of_setup, args):
    # perform some transformation to the image, and then return it as a
    # PIL image or numpy image
    img = do_style_transfer(args["image"])
    # The PIL or numpy image will be automatically converted to a base64
    # encoded data URI string by the @runway.command() decorator.
    return { "image": img }
```

**Parameters**

- **description** (*string, optional*) – A description of this variable and how its used in the model, defaults to None
- **channels** (*int, optional*) – The number of channels in the image, defaults to 3. E.g. an "rgb" image has 3 channels while an "rgba" image has 4.
- **min_width** (*int, optional*) – The minimum width of the image, defaults to None
- **min_height** (*int, optional*) – The minimum height of the image, defaults to None
- **max_width** (*int, optional*) – The maximum width of the image, defaults to None
- **max_height** (*int, optional*) – The maximum height of the image, defaults to None
- **width** (*int, optional*) – The width of the image, defaults to None.
- **height** (*int, optional*) – The height of the image, defaults to None

**class** runway.data_types.**image_bounding_box**(*description=None*)

An bounding box data type, representing a rectangular region in an image. It accepts four normalized floating point numbers [xmin, ymin, xmax, ymax] between 0 and 1, where [xmin, xmax] is the top-left corner of the rectangle and [xmax, ymax] is the bottom-right corner.

```
import runway
from runway.data_types import image, point

@runway.command('detect_face', inputs={'image': image()}, outputs={'face_bbox':␣
↪image_bounding_box})
def detect_faze(model, inputs):
    result = model.run(inputs['image'])
    return {'face_bbox': result}
```

**Parameters** **description** (*string, optional*) – A description of this variable and how it's used in the model, defaults to None

**class** runway.data_types.**image_landmarks**(*length*, *description=None*, *labels=None*, *connections=None*)

An image landmarks data type, representing a fixed-length array of (x, y) coordinates, such as facial landmarks. Each (x, y) coordinate pair in the array should be expressed as normalized image points with values between 0 and 1, inclusive.

```
import runway
from runway.data_types import image, image_landmarks

landmark_names = [
    'nose',
    'leftEye',
```

(continues on next page)

(continued from previous page)

```
    'rightEye',
    'leftEar',
    'rightEar',
    'leftShoulder',
    'rightShoulder',
    'leftElbow',
    'rightElbow',
    'leftWrist',
    'rightWrist',
    'leftHip',
    'rightHip',
    'leftKnee',
    'rightKnee',
    'leftAnkle',
    'rightAnkle'
]

landmark_connections = [
    ['leftHip', 'leftShoulder'],
    ['leftElbow', 'leftShoulder'],
    ['leftElbow', 'leftWrist'],
    ['leftHip', 'leftKnee'],
    ['leftKnee', 'leftAnkle'],
    ['rightHip', 'rightShoulder'],
    ['rightElbow', 'rightShoulder'],
    ['rightElbow', 'rightWrist'],
    ['rightHip', 'rightKnee'],
    ['rightKnee', 'rightAnkle'],
    ['leftShoulder', 'rightShoulder'],
    ['leftHip', 'rightHip']
]

command_outputs = {
    'keypoints': image_landmarks(17, labels=landmark_names, connections=landmark_
↪connections)
}

@runway.command('detect_pose', inputs={'image': image()}, outputs=command_outputs)
def detect_pose(model, inputs):
    pose = model.run(inputs['image'])
    return {'keypoints': pose}
```

**Parameters**

- **length** (*int*) – The number of landmarks associated with this type.
- **labels** (*list, optional*) – Labels associated with each landmark.
- **connections** (*list, optional*) – A list of pairs of logically connected landmarks identified by name, e.g. [[‘left_hip’, ‘left_shoulder], [‘right_hip’, ‘right_shoulder’]]. The names included in connections should correspond to the names provided by the labels property. This property is only used for visualization purposes.
- **description** (*string, optional*) – A description of this variable and how it’s used in the model, defaults to None

**class** runway.data_types.**image_point**(*description=None*)

A point data type representing a specific location in an image. It accepts two normalized floating point numbers

[x, y] between 0 and 1, where [0, 0] represents the top-left corner and [1, 1] the bottom-right corner of an image.

```
import runway
from runway.data_types import image, point

@runway.command('detect_gaze', inputs={'image': image()}, outputs={'gaze_location
↪': image_point()})
def detect_gaze(model, inputs):
    result = model.run(inputs['image'])
    return {'gaze_location': result}
```

**Parameters** **description** (*string, optional*) – A description of this variable and how it's used in the model, defaults to None

**class** runway.data_types.**number**(*description=None*, *default=None*, *step=None*, *min=None*, *max=None*)

A basic number data type. Instantiate this class to create a new runway model variable.

```
import runway
from runway.data_types import number

@runway.setup(options={ "number_of_samples": number })
def setup(opts):
    print("The number of samples is {}".format(opts["number_of_samples"]))
```

**Parameters**

- **description** (*string, optional*) – A description of this variable and how its used in the model, defaults to None
- **default** (*float, optional*) – A default value for this number variable, defaults to 0
- **min** (*float, optional*) – The minimum allowed value of this number type
- **max** (*float, optional*) – The maximum allowed value of this number type
- **step** (*float, optional*) – The step size of this number type. This argument define the minimum change of value associated with this number type. E.g., a step size of *0.1* would allow this data type to take on the values `[0.0, 0.1, 0.2, ..., 1.0]`.

**class** runway.data_types.**segmentation**(*label_to_id=None*, *description=None*, *label_to_color=None*, *default_label=None*, *min_width=None*, *min_height=None*, *max_width=None*, *max_height=None*, *width=None*, *height=None*)

A datatype that represents a pixel-level segmentation of an image. Each pixel is annotated with a label id from 0-255, each corresponding to a different object class.

When used as an input data type, *segmentation* accepts a 1-channel base64-encoded PNG image, where each pixel takes the value of one of the ids defined in *label_to_id*, or a 3-channel base64-encoded PNG colormap image, where each pixel takes the value of one of the colors defined in *label_to_color*.

When used as an output data type, it serializes as a 3-channel base64-encoded PNG image, where each pixel takes the value of one of the colors defined in *label_to_color*.

```
import runway
from runway.data_types import segmentation, image

```

(continues on next page)

(continued from previous page)

```
inputs = {"segmentation_map": segmentation(label_to_id={"background": 0, "person
↪": 1})}
outputs = {"image": image()}
@runway.command("synthesize_pose", inputs=inputs, outputs=outputs)
def synthesize_human_pose(model, args):
    result = model.convert(args["segmentation_map"])
    return { "image": result }
```

**Parameters**

- **description** (*string, optional*) – A description of this variable and how its used in the model, defaults to None
- **label_to_id** (*dict*) – A mapping from labels to pixel values from 0-255 corresponding to those labels
- **default_label** (*string, optional*) – The default label to use when a pixel value not in *label_to_id* is encountered
- **label_to_color** (*dict, optional*) – A mapping from label names to colors to represent those labels
- **min_width** (*int, optional*) – The minimum width of the segmentation image, defaults to None
- **min_height** (*int, optional*) – The minimum height of the segmentation image, defaults to None
- **max_width** (*int, optional*) – The maximum width of the segmentation image, defaults to None
- **max_height** (*int, optional*) – The maximum height of the segmentation image, defaults to None
- **width** (*int, optional*) – The width of the segmentation image, defaults to None.
- **height** (*int, optional*) – The height of the segmentation image, defaults to None

**class** runway.data_types.**text**(*description=None*, *default=''*, *min_length=0*, *max_length=None*)

A basic text data type. Used to represent strings. Instantiate this class to create a new runway model variable.

```
import runway
from runway.data_types import text

@runway.setup(options={ "flavor": text(default="vanilla") })
def setup(opts):
    print("The selected flavor is {}".format(opts["flavor"]))
```

**Parameters**

- **description** (*string, optional*) – A description of this variable and how its used in the model, defaults to None
- **default** (*string, optional*) – The default value for this text variable, defaults to ''
- **min_length** (*int, optional*) – The minimum character length of this text variable, defaults to 0
- **max_length** (*int, optional*) – The maximum character length of this text variable, defaults to None, which allows text to be of any maximum length

**class** runway.data_types.**vector**(*length=None*, *description=None*, *default=None*, *sampling_mean=0*, *sampling_std=1*)

A data type representing a vector of floats.

```
import runway
from runway.data_types import vector, number
import numpy as np

inputs={"length": number(min=1)}
outputs={"vector": vector(length=512)}
@runway.command("random_sample", inputs=inputs, outputs=outputs)
def random_sample(result_of_setup, args):
    vec = vector(length=args["length"])
    rand = np.random.random_sample(args["length"])
    return { "vector": vec.deserialize(rand) }
```

**Parameters**

- **description** (*string, optional*) – A description of this variable and how its used in the model, defaults to None
- **length** (*int, inferred if a default vector is specified*) – The number of elements in the vector
- **sampling_mean** (*float, optional*) – The mean of the sample the vector is drawn from, defaults to 0
- **sampling_std** (*float, optional*) – The standard deviation of the sample the vector is drawn from, defaults to 1

**Raises** ***MissingArgumentError*** – A missing argument error if length is not specified

### 2.2.4 Exceptions

The Runway Model SDK defines several custom exception types. All of these exceptions derive from the base `RunwayError` class, which itself derives from the standard Python `Exception` class. Each `RunwayError` contains error `message` and `code` properties, and a `to_response()` method that converts the exception to a Python dictionary, which can be returned as a JSON HTTP response.

```
try:
    do_something_with_runway()
except RunwayError as e:
    print(e.code, e.message)
    # 500 An unknown error has occurred
    print(e.to_response())
    # { "error": "An unknown error has occurred", "traceback": "..." }
```

#### Reference

**exception** runway.exceptions.**RunwayError**

Bases: `Exception`

A base error class that defines an HTTP error code, an error message, and can be formatted as an object to be returned as JSON in an HTTP request.

**Variables**

- **message** – An error message, set to "An unknown error occurred."
- **code** – An HTTP error code, set to 500

**get_traceback**()
Return a list of lines containing the traceback of the exception currently being handled.

:return A list of lines of the exception traceback :rtype list

**print_exception**()
Print the exception message and traceback to stderr.

**to_response**()
Get information about the error formatted as a dictionary.

**Returns** An object containing "error" and "traceback" keys.

**Return type** dict

**exception** runway.exceptions.**MissingOptionError**(*name*)
Bases: *runway.exceptions.RunwayError*

Thrown by the `@runway.setup()` decorator when a required option value has not been provided by the user.

**Variables**

- **message** – An error message, set to "Missing option: {NAME_OF_MISSING_OPTION}"
- **code** – An HTTP error code, set to 400

**get_traceback**()
Return a list of lines containing the traceback of the exception currently being handled.

:return A list of lines of the exception traceback :rtype list

**print_exception**()
Print the exception message and traceback to stderr.

**to_response**()
Get information about the error formatted as a dictionary.

**Returns** An object containing "error" and "traceback" keys.

**Return type** dict

**exception** runway.exceptions.**MissingInputError**(*name*)
Bases: *runway.exceptions.RunwayError*

Thrown by the `@runway.command()` decorator when a required input value has not been provided by the user.

**Variables**

- **message** – An error message, set to "Missing input: {NAME_OF_MISSING_OPTION}"
- **code** – An HTTP error code, set to 400

**get_traceback**()
Return a list of lines containing the traceback of the exception currently being handled.

:return A list of lines of the exception traceback :rtype list

**print_exception**()
Print the exception message and traceback to stderr.

**to_response**()
Get information about the error formatted as a dictionary.

**Returns** An object containing "error" and "traceback" keys.

**Return type** dict

**exception** runway.exceptions.**InvalidArgumentError**(*name*, *message=None*)
Bases: *runway.exceptions.RunwayError*

An error indicating that an argument is invalid. May be raised by `@runway.setup()` or `@runway.command()` decorated functions if they receive a bad input value.

**Variables**

- **message** – An error message, set to "Invalid argument: {NAME_OF_MISSING_OPTION}"
- **code** – An HTTP error code, set to 400

**get_traceback**()
Return a list of lines containing the traceback of the exception currently being handled.

:return A list of lines of the exception traceback :rtype list

**print_exception**()
Print the exception message and traceback to stderr.

**to_response**()
Get information about the error formatted as a dictionary.

**Returns** An object containing "error" and "traceback" keys.

**Return type** dict

**exception** runway.exceptions.**InferenceError**(*message*)
Bases: *runway.exceptions.RunwayError*

An error thrown if there is an uncaught exception in a function decorated by `@runway.command()`. If this error is thrown, there is an exception in your code ;)

**Variables**

- **message** – A repr() of original exception
- **code** – An HTTP error code, set to 500

**get_traceback**()
Return a list of lines containing the traceback of the exception currently being handled.

:return A list of lines of the exception traceback :rtype list

**print_exception**()
Print the exception message and traceback to stderr.

**to_response**()
Get information about the error formatted as a dictionary.

**Returns** An object containing "error" and "traceback" keys.

**Return type** dict

**exception** runway.exceptions.**UnknownCommandError**(*name*)
Bases: *runway.exceptions.RunwayError*

An error thrown if an HTTP request is made to an endpoint that doesn't exist. E.g. `http://localhost:9000/nothing_here`.

**Variables**

- **message** – An error message, set to "Unknown command: {COMMAND_NAME}"
- **code** – An HTTP error code, set to 404

**get_traceback**()
Return a list of lines containing the traceback of the exception currently being handled.

:return A list of lines of the exception traceback :rtype list

**print_exception**()
Print the exception message and traceback to stderr.

**to_response**()
Get information about the error formatted as a dictionary.

**Returns** An object containing "error" and "traceback" keys.

**Return type** dict

**exception** runway.exceptions.**SetupError**(*message*)
Bases: *runway.exceptions.RunwayError*

An error thrown if there is an uncaught exception in a function decorated by `@runway.setup()`. If this error is thrown, there is an exception in your code ;)

**Variables**

- **message** – A repr() of original exception
- **code** – An HTTP error code, set to 500

**get_traceback**()
Return a list of lines containing the traceback of the exception currently being handled.

:return A list of lines of the exception traceback :rtype list

**print_exception**()
Print the exception message and traceback to stderr.

**to_response**()
Get information about the error formatted as a dictionary.

**Returns** An object containing "error" and "traceback" keys.

**Return type** dict

**exception** runway.exceptions.**MissingArgumentError**(*arg*)
Bases: *runway.exceptions.RunwayError*

An error thrown when a required function argument is not provided.

**Variables** **message** – An error message, set to "Missing argument:

{ARGUMENT_NAME}" :type message: string :ivar code: An HTTP error code, set to 500 :type code: number

**get_traceback**()
Return a list of lines containing the traceback of the exception currently being handled.

:return A list of lines of the exception traceback :rtype list

**print_exception**()
Print the exception message and traceback to stderr.

**to_response**()
Get information about the error formatted as a dictionary.

**Returns** An object containing "error" and "traceback" keys.

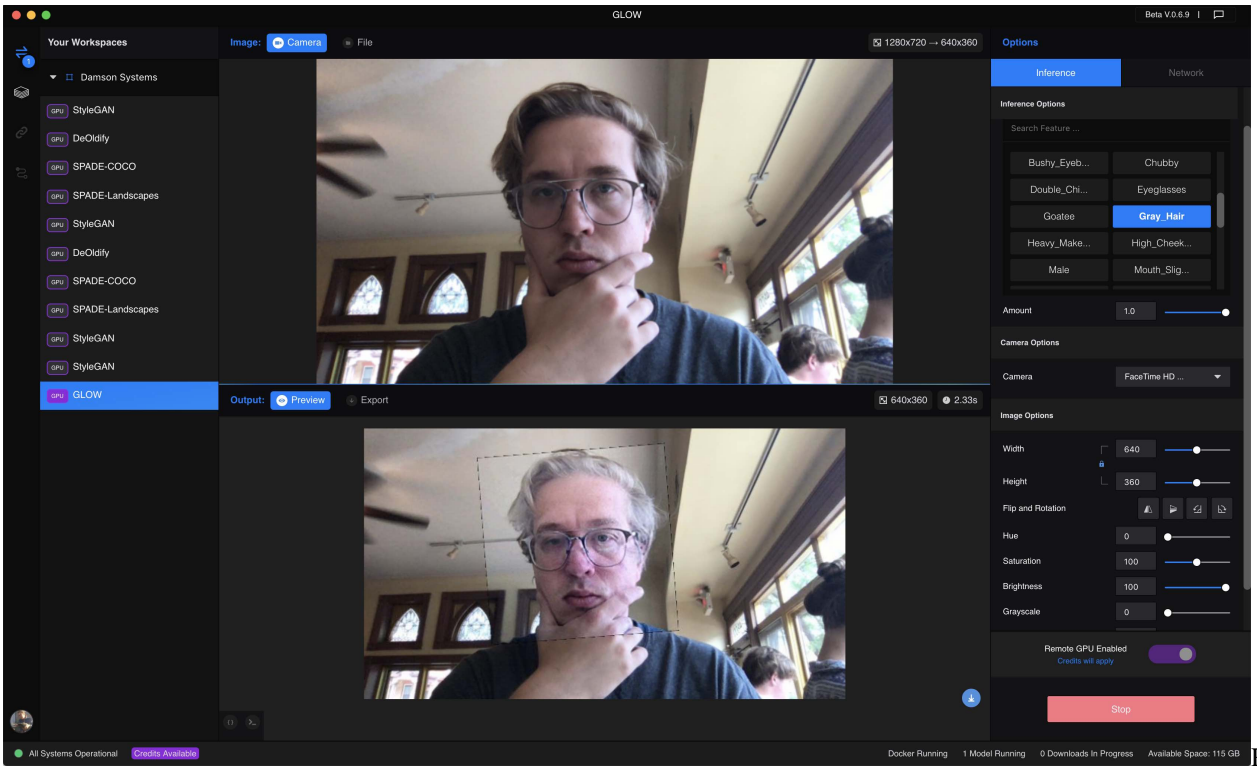
**Return type** dict

### 2.2.5 User Interface Components

One of the main advantages to porting a machine learning model to Runway is that you don't have to spend time designing and building a user interface for users to interact with your model. The Model SDK, in tandem with Runway itself, works to abstract away the details of *how* users get data into and out of a model. As a model porter, you design your model commands, inputs, and outputs using simple Data Types which get automatically transformed into UI components inside the app. You don't have to worry about where the data comes from or where it goes once you process it; Runway takes care of that for you.

This abstraction process also unifies the user experience. Once a user learns how to use their camera to process a live video feed, select an input file from disk, or export the output of a model to CSV, they now know how to do these things with all models; They don't have to re-learn the process for each model.

#### Workspace View



Runway Workspace View

Users interact with your model by adding it to a workspace. From there they are able to chose input and output sources based on the Data Types the model porter specifies in each `@runway.command()` decorator (more on the Runway module here).

The screenshot above depicts an example of OpenAI's Glow model which manipulates facial features of images containing faces. The Glow model that's been ported to Runway expects an `image` as input accompanied by a facial `feature` category like "Young", "Gray_Hair", or "Smiling" and an `amount` that controls the intensity of

the transformation. You will see all three of these inputs listed as "Inference Options" on the right UI control panel. Inference options can be controlled by the user while the model is running (see *Setup Options vs Inference Options* below).

Each model's Inference Options UI panel is automatically generated based on the code in its `runway_model.py` file. The model porter controls which types of UI components are used for each model using only the `inputs` and `outputs` keyword arguments to each `@runway.command()`. Below is a toy example of what the `runway_model.py` code that generates the UI components looks like for Glow. This code is used for illustrative purposes. Click here to see the actual code for the Glow model in Runway.

```
import runway
from runway import image, category, number
from fake_glow_code import manipulate_image_with_glow

# Facial feature manipulations that can be applied to the input image. Truncated for␣
↪brevity.
features = '5_o_Clock_Shadow Arched_Eyebrows Attractive Bags_Under_Eyes Bald Bangs'
features = features.split()

manipulate_command_inputs = {
  'image': image(description='The input image containing a face to transform.'),
  'feature': category(choices=features, default=features[2], description='Facial␣
↪feature to apply.'),
  'amount': number(default=0.5, min=0, max=1, step=0.1, description='The␣
↪transformation intensity.')
}

manipulate_command_outputs = {
  'output': image(description='The output image containing the transformed face.')
}

manipulate_command_desc = 'Manipulate an image of a face by applying facial feature␣
↪transformations.'

@runway.command('manipulate',
                inputs=manipulate_command_inputs,
                outputs=manipulate_command_outputs,
                description=manipulate_command_desc)
def manipulate(model, args):
  return {
    'output': manipulate_image_with_glow(args)
  }
```

### Input Sources and Output Destinations

Model porters define the types of input a model receives and the types of output a model produces, but they **do not** define the sources of those inputs or the destinations for those outputs. When a user runs a model that accepts an image as input and produces another image as output **the user** choses where that image comes from before it's sent to the model and where it goes once the model produces a new image as output. This concept applies to all data types, not just images.

### Input Sources

- **Camera**: A webcam or USB camera can be used as an input source for all models that accept an image as input.
- **Text**: A simple text area box for typing or pasting text based input.

- **Vector**: A fixed-width z-vector of floats used as the seed for some generative models like StyleGAN
- **Segmentation**: A drawing tool for creating images composed of objects that represent semantic classes. See SPADE-COCO for an example of this input source.
- **File**: File is unique because it's used to load files from disk in a uniform way across all Data Types. This input source is used to feed a model input from your filesystem.
- **From Network**: This input source is also unique in that it appears whenever you interact with a model via the network. This input source is available for all models but only appears once model a model has been triggered by a network request.

---

**Note:** Both the **File** and **From Network** input sources are unique in that they aren't made available to users as a result something the model porter defines in a `@runway.command()` decorator, but rather they are **always** available to users to load their own data of arbitrary type. **From Network** will only appear once a model has been triggered by a network request.

---

### Output Destinations

- **Preview**: A basic preview window for users to view the output of the model inside of Runway. **Preview** renders output differently depending on the the output type: For instance, it will render an image if a model outputs an image and text if a model outputs text.
- **Export**: Save model output to disk as an exported video, image sequence, or structured text file like CSV or JSON depending on the output data type.

### Model Setup Options vs Inference Options

The data types specified in the `options` keyword argument of `@runway.setup()` and the `inputs` keyword arguments of `@runway.command()` functions are handled somewhat differently by Runway. The former defines options that a user configures **before** the model is started, while the later can be adjusted at any time while the model is running.

```
import runway
from runway import category, text
from elsewhere import model

setup_options={
  'architecture': category(choices=['Artistic', 'Stable', 'Video'])
}
@runway.setup(options=setup_options)
def setup(opts):
  return model(opts['architecture'])

generate_inputs={
  'render_factor': number(min=1, max=50, default=35)
}

generate_outputs={
  'output': image
}
@runway.command('generate',inputs=generate_inputs, outputs=generate_outputs)
def generate(model, opts):
  return model.generate_image(opts['render_factor'])
```

As a model porter you must decide which of the options that are configurable by the user should be setup options and which should be inference options. A good rule of thumb is that if an option is needed to load or initialize a model it should be a setup option. Otherwise it should be an inference option.

---

**Note:** The same data types and UI components are used in both Model Setup Options and Inference Options. The only difference is that Setup Model options can only be selected before you run the model.

---

### UI Components

Only a limited set of all Data Types generate UI components at this time. We are working quickly to render all data types in the app, but we ask for your patience until this task is complete. If you have any questions about data types, UI components, or the Model SDK in general feel free to bring them up in the `#model-sdk` channel in the public Runway slack.

### Support/Compatibility Matrix

* Denotes this feature is coming soon!

| Data Type | Setup Option | Command Input | Command Output | Notes |
|---|---|---|---|---|
| `image` | No | **Yes** | **Yes** | The `image` type is also used to process video sequences. |
| `text` | No | **Yes** | **Yes** | |
| `boolean` | **Yes** | **Yes** | No* | Boolean data types used as input appear on the model options UI on the right side of the workspace view. |
| `category` | **Yes** | **Yes** | No* | Category data types used as input appear on the model options UI on the right side of the workspace view. If you are looking to output a category, instead output the category value as a string using the `text` type for now. |
| `number` | **Yes** | **Yes** | No | Number data types used as input appear on the model options UI on the right side of the workspace view. If you are looking to output a number, instead output a number as a string using the `text` type for now. |
| `vector` | No | **Yes** | No | The vector input uses a grid of images. More data types will be supported soon. See StyleGAN for an example. |
| `segmentation` | No | **Yes** | **Yes** | Use a regular image as output for now. |
| `file^` | **Yes** | No | No | ^ The file input should not be used as an input or output type for ‘‘@runway.command()‘‘s. Instead the file UI component will be rendered as an optional input source for all types. |
| `array` | No | No* | No* | If you are looking to output an array of strings, join them using a `,` ``` and return a ```text` type for now. |
| `any` | No | No | No | The any data type is meant to be extensible and used primarily via the Network. |

### Examples

Each data type is rendered differently in the Runway workspace UI. We'll have a look at what each input data type in detail below.

- *Image*

- *Text*
- *Vector*
- *Segmentation*
- *Number*
- *Boolean*

### Image

```
import runway
from runway import image
# this is a fake import for demonstration purposes only
from elsewhere import process_input_image

# The "image" and "output" keys in the input and output dictionaries are arbitrarily␣
↪named
@runway.command('example', inputs={ 'image': image }, outputs={ 'output': image })
def example(model, args):
  return process_input_image(args['image'])
```

The above code will generate an `image` UI component. The user will then be able to choose either a camera or an image file as the source of the input.


Each input image type also creates an Image Options UI component on the right panel of the workspace view. This panel gives the users the option to resize and manipulate the input image before its sent to the model. The model porter does not have control over these parameters as they are up to the user to chose and will have already been applied to the image by the time it reaches the Model SDK code.

### Text

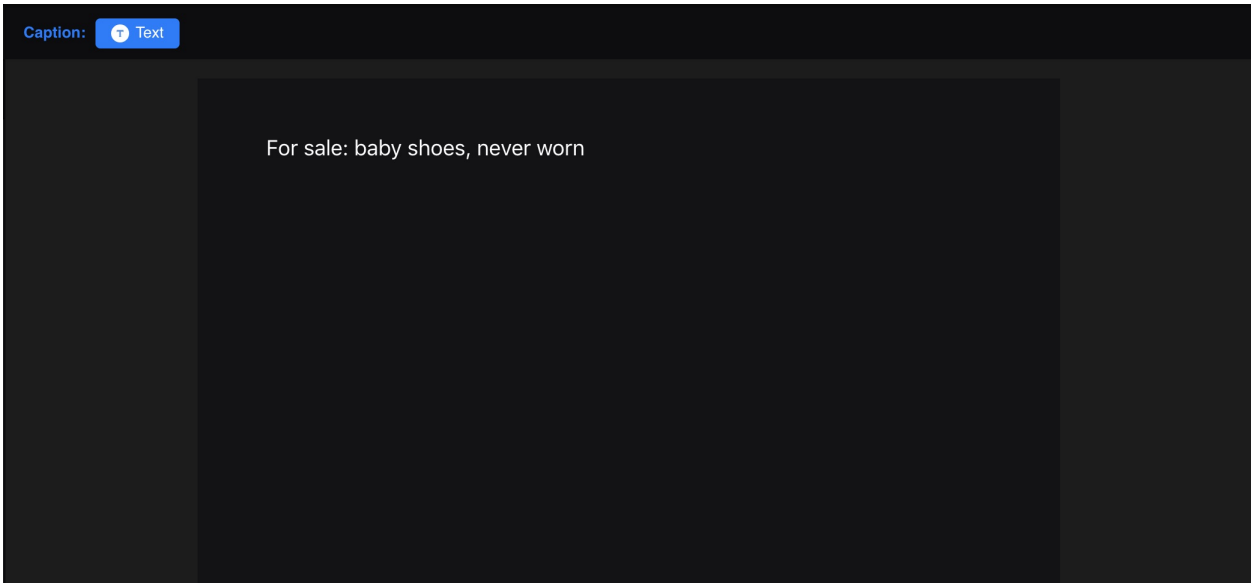
```
import runway
from runway import text
```

(continues on next page)

(continued from previous page)

```
# The "caption" and "informalized" keys in the input and output dictionaries are␣
↪arbitrarily named
@runway.command('informalize', inputs={ 'caption': text }, outputs={ 'informalized':␣
↪text })
def informalize(model, args):
  return args['caption'].lower()
```

The above code will generate a `text` UI component. The user will be able to type a string of text into the text box, and that text will be used as the `caption` input to the model.

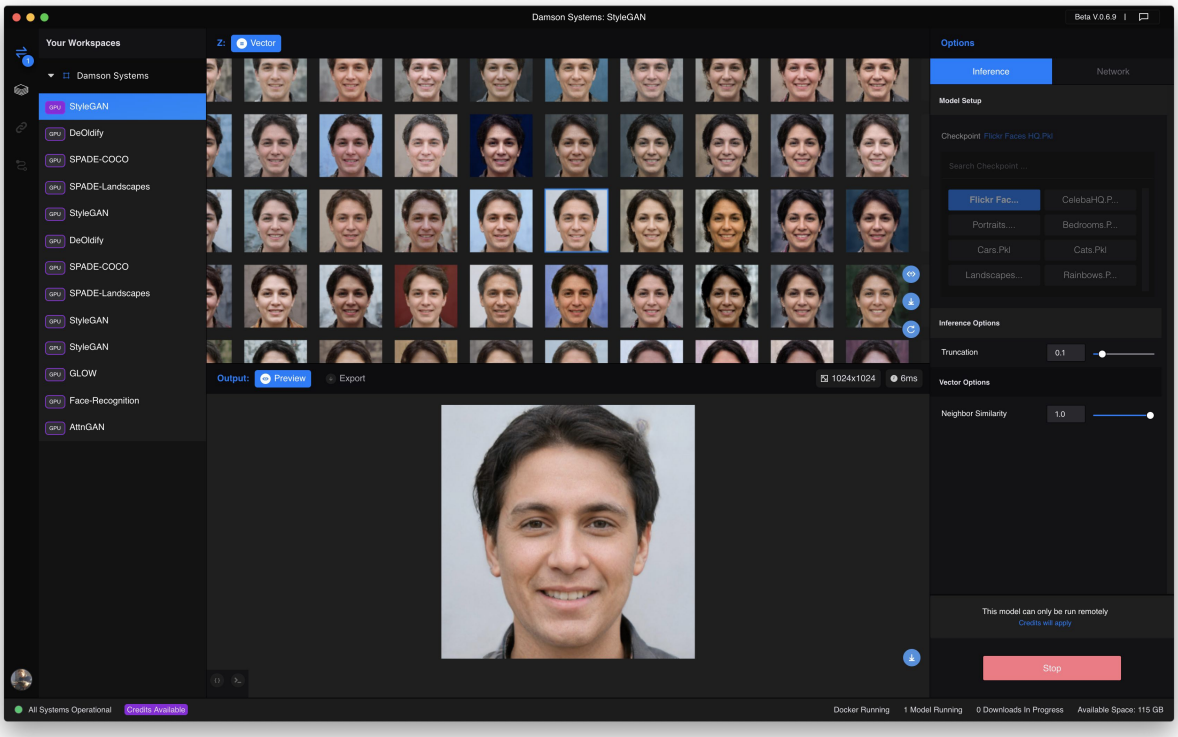



### Vector

The code below illustrates how to use the `vector` data type to explore a hyperparameter space by incrementally adding noise to fixed-width z-vector that's used as input to a generative model. This data type is a bit tricky to understand if you don't have much experience with generative models, but have a look at StyleGAN for an example of its use.

```
import runway
from runway import vector, image
# this is a fake import for demonstration purposes only
from elsewhere import sample_image_from_z_vector

# The z" and "output" keys in the input and output dictionaries are arbitrarily named
@runway.command('sample', inputs={ 'z': vector }, outputs={ 'output': image })
def sample(model, args):
  return sample_image_from_z_vector(args['z'])
```




## Segmentation

The segmentation data type is used to define per-pixel labels to an image. Each pixel is annotated with a label id from 0-255, each corresponding to a different object class. Models like NVIDIA's SPADE use this data type to generate photorealistic output images using input semantic segmentation map images of only a handful of colors.

```
import runway
from runway import image, segmentation
# this is a fake import for demonstration purposes only
from elsewhere import generate_image_from_semantic_map

label_to_id = {
    'unlabeled': 0,
    'grass': 124,
    'sky': 156,
    'clouds': 105,
    'sea': 154,
    'river': 148,
    'tree': 169,
    'mountain': 134
}

label_to_color = {
    'unlabeled': (0, 0, 0),
    'grass': (29, 195, 49),
    'sky': (95, 219, 255),
    'clouds': (170, 170, 170),
    'sea': (54, 62, 167),
```

(continues on next page)

(continued from previous page)

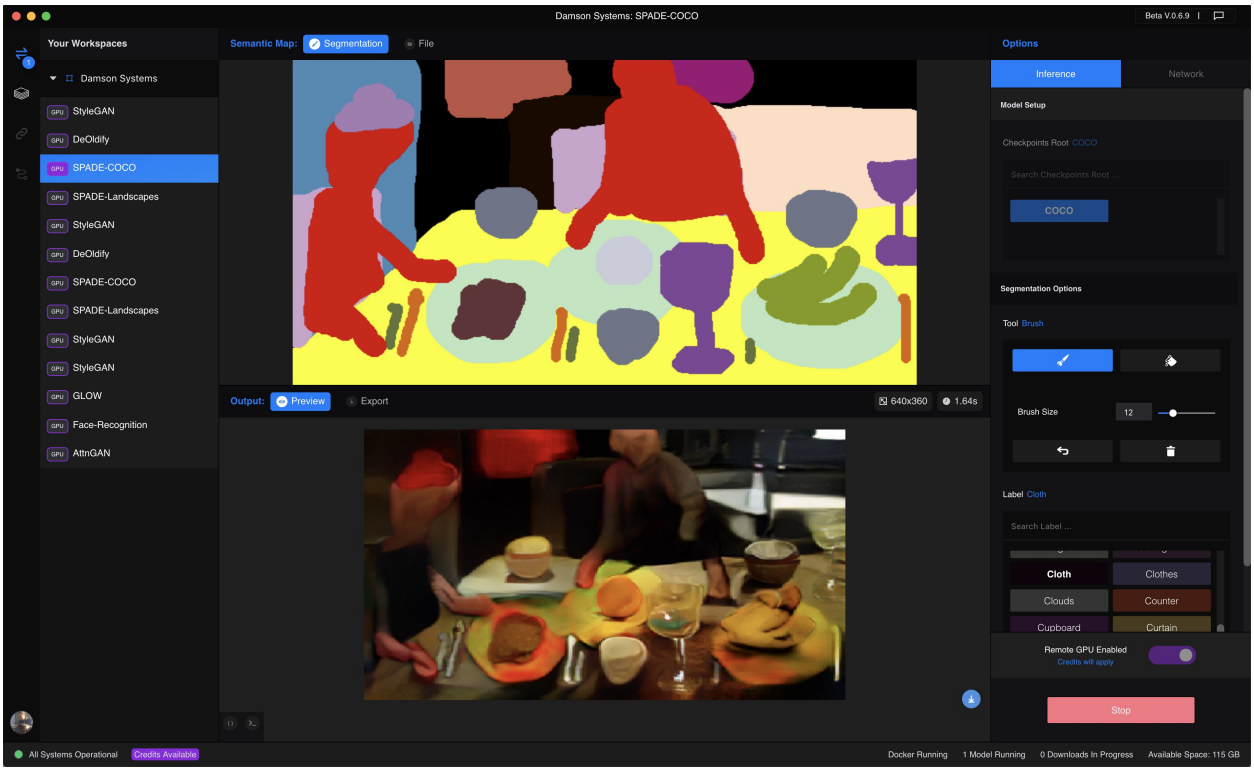
```
    'river': (0, 57, 150),
    'tree': (140, 104, 47),
    'mountain': (60, 55, 50)
}

convert_inputs = {
    'semantic_map': segmentation(
      label_to_id=label_to_id,
      label_to_color=label_to_color,
      default_label='unlabeled',
      width=640,
      height=360
    )
}

convert_outputs = {
    'output': image
}

@runway.command('convert', inputs=convert_inputs, outputs=convert_outputs)
def convert(model, args):
    return generate_image_from_semantic_map(args['semantic_map'])
```

Using the `segmentation` data type as input to a command generates a UI component that allows a user to draw directly on a canvas using primitive painting tools and colors that map to the semantic map labels.



### Number

A number used as input to a model command creates a number slider on the right Model Options UI panel. This data type is often used to pass an adjustable scalar value to a model at runtime, as is illustrated below.

```
import runway
from runway import number, image
# this is a fake import for demonstration purposes only
from elsewhere import stylize

stylize_inputs = {
  'input_image': image,
  'style_image': image,
  'amount': number(min=0, max=100, default=50)
}

stylize_outputs = {
  'stylized_image': image
}
@runway.command('stylize_image',
                inputs=stylize_inputs,
                outputs=stylize_outputs,
                description='Stylize an image by a certain amount using a style image
↪')
def stylize_image(model, args):
  return stylize(args['input_image'], args['style_image'], args['amount'])
```

Here is an example of the number slider produced by the code above.

### Boolean

Using the boolean data type as input to a model command generates a simple switch button on the right Model Options UI panel.

```
import runway
from runway import boolean, text
# this is a fake import for demonstration purposes only
from elsewhere import translate, tokenize

translate_inputs = {
  'english_input': image,
  'tokenize': boolean(default=True),
}

translate_outputs = {
  'french_output': text
}
@runway.command('translate',
                inputs=translate_inputs,
                outputs=translate_outputs,
                description='Translate sentences from English to French')
def translate(model, args):
  english = args['english_input']
  if args['tokenize']:
    english = tokenize(english)
  return translate(english, 'french')
```

Here is an example of the number slider produced by the code above.

### 2.2.6 Example Models

In order to better understand the process of porting a model to Runway, we recommend checking out the source code for some of the models that have already been ported. All models published by the runway organization are open source, as well as many of the models contributed by our community.

- Image-Super-Resolution: A very simple image upscaling model that gives a good overview of the role of `runway_model.py`.
- StyleGAN: A good example of loading a model checkpoint as well as using the vector data type.
- SPADE-COCO: A good example of the segmentation data type.
- DeepLabV3: A good example of multiple `@command()` functions and conditional build steps depending on GPU and CPU build environments. Also a very simple model to get started with.
- Places365: A good example of a basic image classification task and use of the text data type for output.
- 3DDFA: A good example of dealing with 3D data as images. We plan to add more features for handling true 3D data.

# PYTHON MODULE INDEX

## r

# INDEX

## R



## S



## T



## U



## V