# IEEE Computer Society Gym Cheat Sheet

Gym is a toolkit for developing and comparing reinforcement learning algorithms. It makes no assumptions about the structure of your agent, and is compatible with any numerical computation library, such as TensorFlow or Theano.

## Basics

There are two basic concepts in reinforcement learning: the environment (namely, the outside world) and the agent (namely, the algorithm you are writing). The agent sends actions to the environment, and the environment replies with observations and rewards (that is, a score).

The core gym interface is Env, which is the unified environment interface. There is no interface for agents; that part is left to you. The following are the Env methods you should know:

`reset`(self): Reset the environment's state. Returns observation.

`step`(self, action): Step the environment by one timestep. Returns observation, reward, done, info.

`render`(self, mode='human'): Render one frame of the environment. The default mode will do something human friendly, such as pop up a window.

## Installation

To get started, you'll need to have Python 3.5+ installed. Simply install gym using pip:

```
pip install gym
```

And you're good to go!

## Environments

Here's a bare minimum example of getting something running. This will run an instance of the CartPole-v0 environment for 1000 timesteps, rendering the environment at each step. You should see a window pop up rendering the classic cart-pole problem:

```
import gym
env = gym.make('CartPole-v0')
env.reset()
for _ in range(1000):
    env.render()
    env.step(env.action_space.sample()) # take a random action
env.close()
```

If you'd like to see some other environments in action, try replacing CartPole-v0 above with something like MountainCar-v0, MsPacman-v0 (requires the Atari dependency), or Hopper-v1 (requires the MuJoCo dependencies). Environments all descend from the Env base class.

## Observations

If we ever want to do better than take random actions at each step, it'd probably be good to actually know what our actions are doing to the environment.

The environment's step function returns exactly what we need. In fact, step returns four values. These are:

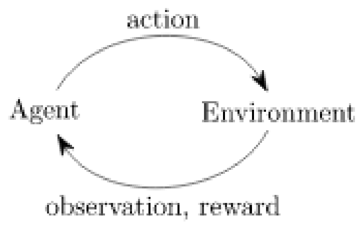
`observation` (object): an environment-specific object representing your observation of the environment. For example, pixel data from a camera, joint angles and joint velocities of a robot, or the board state in a board game.

`reward` (float): amount of reward achieved by the previous action. The scale varies between environments, but the goal is always to increase your total reward.

`done` (boolean): whether it's time to reset the environment again. Most (but not all) tasks are divided up into well-defined episodes, and done being True indicates the episode has terminated. (For example, perhaps the pole tipped too far, or you lost your last life.)

`info` (dict): diagnostic information useful for debugging. It can sometimes be useful for learning (for example, it might contain the raw probabilities behind the environment's last state change). However, official evaluations of your agent are not allowed to use this for learning.

This is just an implementation of the classic "agent-environment loop". Each timestep, the agent chooses an action, and the environment returns an observation and a reward.



The process gets started by calling `reset`(), which returns an initial observation. So a more proper way of writing the previous code would be to respect the done flag:

```
import gym
env = gym.make('CartPole-v0')
for i_episode in range(20):
    observation = env.reset()
    for t in range(100):
        env.render()
        print(observation)
        action = env.action_space.sample()
        observation, reward, done, info = env.step(action)
        if done:
            print("Episode finished after {} timesteps".format(t+1))
            break
env.close()
```

## Spaces

In the examples above, we've been sampling random actions from the environment's action space. But what actually are those actions? Every environment comes with an action_space and an observation_space. These attributes are of type Space, and they describe the format of valid actions and observations:

```
import gym
env = gym.make('CartPole-v0')
print(env.action_space)
#> Discrete(2)
print(env.observation_space)
#> Box(4,)
```

The Discrete space allows a fixed range of non-negative numbers, so in this case valid actions are either 0 or 1. The Box space represents an n-dimensional box, so valid observations will be an array of 4 numbers. We can also check the Box's bounds:

```
print(env.observation_space.high)
#> array([ 2.4       ,         inf,  0.20943951,         inf])
print(env.observation_space.low)
#> array([-2.4       ,        -inf, -0.20943951,        -inf])
```

This introspection can be helpful to write generic code that works for many different environments. Box and Discrete are the most common Spaces. You can sample from a Space or check that something belongs to it:

```
from gym import spaces
```

```
space = spaces.Discrete(8) # Set with 8 elements {0, 1, 2, ..., 7}
x = space.sample()
assert space.contains(x)
assert space.n == 8
```

For CartPole-v0 one of the actions applies force to the left, and one of them applies force to the right. (Can you figure out which is which?)

Fortunately, the better your learning algorithm, the less you'll have to try to interpret these numbers yourself.

## The registry

gym's main purpose is to provide a large collection of environments that expose a common interface and are versioned to allow for comparisons. To list the environments available in your installation, just ask gym.envs.registry:

```
from gym import envs
print(envs.registry.all())
#> [EnvSpec(DoubleDunk-v0), EnvSpec(InvertedDoublePendulum-v0),
EnvSpec(BeamRider-v0), EnvSpec(Phoenix-ram-v0), EnvSpec(Asterix-v0),
EnvSpec(TimePilot-v0), EnvSpec(Alien-v0), EnvSpec(Robotank-ram-v0),
EnvSpec(CartPole-v0), EnvSpec(Berzerk-v0), EnvSpec(Berzerk-ram-v0),
EnvSpec(Gopher-ram-v0), ...
```

This will give you a list of EnvSpec objects. These define parameters for a particular task, including the number of trials to run and the maximum number of steps. For example, EnvSpec(Hopper-v1) defines an environment where the goal is to get a 2D simulated robot to hop; EnvSpec(Go9x9-v0) defines a Go game on a 9x9 board.

These environment IDs are treated as opaque strings. In order to ensure valid comparisons for the future, environments will never be changed in a fashion that affects performance, only replaced by newer versions. We currently suffix each environment with a v0 so that future replacements can naturally be called v1, v2, etc.

It's very easy to add your own environments to the registry, and thus make them available for gym.make(): just register() them at load time.