Let’s Unify.ai - Ivy

“Simplified Guide to Ivy for New Contributors”

By Ambroise L.

ML Research Engineer at Ivy

August 2023

University at Albany, NY

College of Engineering and Applied Science

Computer Science Department

Tompkins Cortland Community College, Dryden, NY

Department of Engineering Science

We Would Recommend Reading Through Our Docs and Trying to Gain a Good Understanding of Ivy Before Starting. Then, Can You Guide Me to the Right Ivy Documents to Read for this Purpose?

Sure, I can guide you to the relevant Ivy documents that can help you gain a deeper understanding of the framework. Here are some key sections you should focus on:

1. **Design:** This section provides a user-focused guide about the architecture and the building blocks of Ivy. It's a great place to start to understand how Ivy is structured and how it works. **Link** "Starting with our higher-level design section, Ivy can serve two distinct purposes:
    1. Enabling automatic code conversions between frameworks.
    2. Serving as a new ML framework with multi-framework support." Link **to Design**.
2. **Deep Dive**: This section is more targeted at people who would like to dive deeper into how Ivy actually works under the hood. It covers various aspects of Ivy, including navigating the code, function types, superset behavior, backend setting, arrays, containers, data types, devices, and inplace updates. **Link to Deep Dive**
3. **Ivy as a Framework**: This section explains how Ivy can be used as a fully-fledged ML framework. It covers the ivy.Container class, the ivy.Array class, and the stateful API. **Link to Ivy as a Framework**
4. **Ivy Frontends**: This section explains the role of the backend-specific frontends in Ivy, and how these enable automatic code conversions between different ML frameworks. **Link to Ivy Frontends**
5. **Contributing**: If you're interested in contributing to Ivy, this section provides a guide on how to get started, including setting up your environment, creating issues and pull requests, building the docs, and more. **Link to Contributing**

Remember, the goal is to understand how Ivy works and how you can use it to build and train ML models. Don't hesitate to reach out to the Ivy community if you have any questions!

Sources:

unify.ai/docs

unify.ai/docs

unify.ai/docs

unify.ai/docs

unify.ai/docs

1. Ivy's Design

"Starting with our higher-level design section, Ivy can serve two distinct purposes:

1. Enabling automatic code conversions between frameworks.
2. Serving as a new ML framework with multi-framework support.

It's intriguing to see how Ivy serves a dual purpose in automatic code conversion and functioning as a versatile (the ability to adapt and perform well in various different situations or roles) ML framework with support for multiple frameworks. This design approach seems to acknowledge the importance of interoperability and flexibility in the field of machine learning. The ability to seamlessly convert code between different frameworks can greatly enhance the efficiency of ML development, while simultaneously acting as a new framework showcases Ivy's ambition to provide a robust and adaptable environment for researchers and developers. This approach could potentially lead to more streamlined workflows and broader accessibility in the ever-evolving landscape of machine learning.

The Ivy codebase's organization into three distinct categories, denoted as (a), (b), and (c), is a structured approach that allows for a clear breakdown of its components. These categories further comprise a total of eight well-defined submodules, each serving specific functionalities. The eight submodules include the Ivy API, Backend Handler, Backend API, Ivy Array, Ivy Container, Ivy Stateful API, and the Frontend API. These submodules are intelligently allocated among the three categories as follows:

A. Building Blocks:

- Back-end functional APIs ✅
- Ivy functional API ✅
- Framework Handler ✅
- Ivy Compiler 🚧

B. Ivy as a Transpiler:

- Front-end functional APIs 🚧

C. Ivy as a Framework:

- Ivy Stateful API ✅
- Ivy Container ✅
- Ivy Array ✅

This organization demonstrates a comprehensive approach to structuring the Ivy codebase, ensuring that each module's purpose and contribution are clearly outlined. The use of visual indicators like ✅ and 🚧 adds a helpful touch for indicating the completeness and progress of each submodule.

A. Building Blocks

A. 1. Backend Functional APIs ✅

The first important point to make is that, Ivy does not implement its own C++ or CUDA backend. Instead, Ivy **wraps** the functional APIs of existing frameworks, bringing them into syntactic and semantic alignment. Let's take the function [ivy.stack()]() as an example below.

Here is the code example for you:

```
# ivy/functional/backends/numpy/manipulation.py:
def stack(
    arrays: Union[Tuple[np.ndarray], List[np.ndarray]],
    /,
    *,
    axis: int = 0,
    out: Optional[np.ndarray] = None,
) -> np.ndarray:
    return np.stack(arrays, axis, out=out)

stack.support_native_out = True
```

This code is part of the Ivy ML framework's backend for Numpy. It defines a stack function that joins a sequence of arrays along a new axis. The stack function supports native output, as indicated by the stack.support_native_out = True line.

There are separate backend modules for JAX, TensorFlow, PyTorch, and NumPy, and so we implement the stack method once for each backend, each in separate backend files like so:

There were no changes required for this function, however NumPy and PyTorch both had to be marked as supporting the [out argument]() (Most functions in Ivy support inplace updates via the inclusion of a keyword-only out argument. This enables users to specify the array to which they would like the output of a function to be written.) natively.

For the complicated functions such as logspace method we need to do more than simply wrap and maybe change the name: What does logspace method do?

Here is the code example for you:

```
from typing import Union, Optional
import tensorflow as tf
import ivy
```

```
def logspace(
    start: Union[tf.Tensor, tf.Variable, int],
    stop: Union[tf.Tensor, tf.Variable, int],
    num: int,
    base: float = 10.0,
    axis: Optional[int] = None,
    *,
    dtype: tf.DType,
    device: str,
) -> Union[tf.Tensor, tf.Variable]:
    power_seq = ivy.linspace(start, stop, num, axis, dtype=dtype, device=device)
    return base**power_seq
```

This function generates a sequence of numbers that are evenly spaced on a log scale. The start and stop arguments define the range of the sequence, and num specifies the number of steps. The base argument is used for the logarithmic scale, and the dtype and device arguments specify the data type and the device where the tensor will be created, respectively.

This function is part of the TensorFlow backend for the Ivy machine learning framework. It's used to provide a unified API for different backends, including TensorFlow, PyTorch, JAX, and NumPy.

Source: [Ivy Documentation]

Here are the parameters it accepts:

- **start**: First value in the range in log space. base ** start is the starting value in the sequence. Can be an array or a float.
- **stop**: Last value in the range in log space. base ** stop is the final value in the sequence. Can be an array or a float.
- **num**: Number of values to generate.
- **base**: The base of the log space. Default is 10.0
- **axis**: Axis along which the operation is performed. Relevant only if start or stop are array-like. Default is 0.
- **endpoint**: If True, stop is the last sample. Otherwise, it is not included. Default is True.
- **dtype**: The data type of the output tensor. If None, the dtype of on_value is used or if that is None, the dtype of off_value is used, or if that is None, defaults to float32. Default is None.
- **device**: device on which to create the array ‘cuda:0’, ‘cuda:1’, ‘cpu’ etc. Default is None.
- **out**: optional output array, for writing the result to. It must have a shape that the inputs broadcast to. Default is None.

The method returns a tensor of evenly-spaced values in log space.

Here is an example of its usage:

```
>>> print(ivy.logspace(1, 2, 4))
ivy.array([ 10., 21.5443469, 46.41588834, 100.])
```

A. 2. Ivy Functional APIs ✅

Calling the different backend files explicitly would work okay, but it would mean we need to import ivy.functional.backends.torch as ivy to use a PyTorch backend or import ivy.functional.backends.tensorflow as ivy to use a TensorFlow backend. Instead, we allow these backends to be bound to the single shared "namespace ivy". The backend can then be changed by calling ivy.set_backend('torch') for example.

Instead, we allow these backends to be bound to the single shared namespace ivy. What is namespace ivy?

The namespace ivy in the context of Ivy, the ML framework, is a shared namespace that allows different backend files to be bound to it. This design allows users to switch between different ML backends such as PyTorch, TensorFlow, JAX, and Numpy without having to import different backend files explicitly.

For instance, instead of importing a PyTorch backend with import ivy.functional.backends.torch as ivy or a TensorFlow backend with import ivy.functional.backends.tensorflow as ivy, users can simply use the ivy namespace and change the backend by calling ivy.set_backend('torch') or ivy.set_backend('tensorflow').

This design also allows Ivy to implicitly set numpy as the default backend or operate with the backend corresponding to the specified data inputs until the user explicitly sets a different backend.

Here's an example of how to use the ivy namespace to switch between backends:

```
# implicit
import ivy
x = ivy.array([1, 2, 3])
(type(ivy.to_native(x)))
# -> <class 'numpy.ndarray'>

import torch
t = torch.tensor([23,42, -1])
type(ivy.to_native(ivy.sum(t)))
# -> <class 'torch.Tensor'>

# explicit
import ivy
ivy.set_backend("jax")
```

```
z = ivy.array([1, 2, 3]))
type(ivy.to_native(z))
# -> <class 'jaxlib.xla_extension.DeviceArray'>
```

For more details, you can refer to the [Building Blocks > Ivy Functional API]() section of the Ivy documentation.

In Ivy's framework, there's a path called “ivy.functional.ivy.” This path is where all the explanations about how doc strings work and the information about the argument typing reside. This is especially for the functional part of Ivy that users (people) use.

For instance, there's a function called “prod()” that we can see below:

The ivy.prod() function in Ivy is used to calculate the product of elements in an input array. The function is part of Ivy's functional API and can be used with any of the supported backends (JAX, TensorFlow, PyTorch, and Numpy).

Here is the function signature:

```
def prod(
    x: Union[ivy.Array, ivy.NativeArray],
    *,
    axis: Optional[Union[int, Sequence[int]]] = None,
    dtype: Optional[Union[ivy.Dtype, ivy.NativeDtype]] = None,
    keepdims: bool = False,
    out: Optional[ivy.Array] = None,
) -> ivy.Array:
```

The parameters are:

- x: The input array.
- axis: The axis or axes along which the product must be computed. By default, the product is computed over the entire array.
- dtype: The data type of the returned array.
- keepdims: If True, the reduced axes (dimensions) are included in the result as singleton dimensions.
- out: An optional output array for writing the result to.

Here is an example of how to use the function:

```
import ivy

x = ivy.array([1, 2, 3])
z = ivy.prod(x)
print(z)  # Outputs: ivy.array(6)
```

You can also use ivy.prod() with Ivy's container data structure:

```
x = ivy.Container(a=ivy.array([0.1, 1.1]), b=ivy.array([0.1, 1.1, 2.1]))
y = x.prod(keepdims=True)
print(y)
# Outputs: {a: ivy.array([0.11000001]), b: ivy.array([0.23100001])}
```

For more details, you can refer to the Ivy documentation.

A. 3. Backend Functional APIs ✅

#current_backend

The code responsible for both choosing (setting) and changing (unsetting) the backend is stored within the submodule located at ivy/utils/backend/handler.py. The main function that you interact with, which is visible from the front end, is called ivy.current_backend(). Below, we show what the code inside this function looks like.

Can you explain the ivy.current_backend() function?

The ivy.current_backend() function is used to return the current backend that Ivy is using. The backend is the underlying machine learning framework that Ivy uses to perform its operations, such as TensorFlow, PyTorch, JAX, or Numpy.

The function works by first checking if a global backend has been set using the ivy.set_backend() function. If a global backend has been set, this backend is returned. If no global backend has been set, the function tries to “infer the backend” from the input arguments.

Here is the code for the ivy.current_backend() function for you:

```
# ivy/backend_handler.py
 def current_backend(*args, **kwargs):
    global implicit_backend
    # if a global backend has been set with set_backend then this will be returned
    if backend_stack:
        f = backend_stack[-1]
        if verbosity.level > 0:
            verbosity.cprint("Using backend from stack: {}".format(f))
        return f

    # if no global backend exists, we try to infer the backend from the arguments
    f = _determine_backend_from_args(list(args) + list(kwargs.values()))
    if f is not None:
        implicit_backend = f.current_backend_str()
        return f
    if verbosity.level > 0:
        verbosity.cprint("Using backend from type: {}".format(f))
```

```
    return importlib.import_module(_backend_dict[implicit_backend])
```

For example, if no global backend is set, then the backend is inferred from the arguments --

Then, what do you refer to by "infer the backend?"

"Infer the backend" refers to the process of determining which backend framework to use based on the input arguments provided to the function. If no global backend has been set using ivy.set_backend(), Ivy will check the types of the input arguments to determine the appropriate backend.

For example, if you pass a TensorFlow tensor as an argument to an Ivy function and no global backend has been set, Ivy will infer that the TensorFlow backend should be used for that function call. Similarly, if a PyTorch tensor is passed, Ivy will infer that the PyTorch backend should be used.

Here is an example from the documentation:

```
>>> import numpy as np
>>> x = np.array([2.0])
>>> print(ivy.current_backend(x))
<module 'ivy.functional.backends.numpy' from
'/ivy/ivy/functional/backends/numpy/__init__.py'> # noqa
```

In this example, a Numpy array is passed as an argument, so Ivy infers that the Numpy backend should be used.

This feature allows Ivy to be flexible and adaptable to different backend frameworks, making it easier for users to switch between different ML frameworks like TensorFlow, PyTorch, JAX, and Numpy.

The global backend set in ivy.set_backend() has priority over any arguments passed to ivy.current_backend():

```
>>> import numpy as np
>>> ivy.set_backend("jax")
>>> x = np.array([2.0])
>>> print(ivy.current_backend(x))
<module 'ivy.functional.backends.jax' from '/ivy/ivy/functional/backends/jax/__init__.py'> #
noqa
```

This feature allows Ivy to be flexible and adaptable to different backend frameworks, making it easier for users to switch between different ML frameworks like TensorFlow, PyTorch, JAX, and Numpy.

You can find more information about this function in the Ivy documentation.

#set_backend

We try to clarify this below: The functions in this returned module are populated by iterating through the global ivy.__dict__ (or a non-global copy of ivy.__dict__ if non-globally-set), and overwriting every function which is also directly implemented in the backend-specific namespace. The following is a slightly simplified version of this code for illustration, which updates the global ivy.__dict__ directly:

More explanation about it:

In Ivy, the backend (i.e., the underlying machine learning framework like TensorFlow, PyTorch, etc.) can be set using the set_backend function. When this function is called, it updates the ivy.__dict__ object, which is a dictionary that contains all the functions available in Ivy.

The set_backend function works by iterating through the ivy.__dict__ object and overwriting every function that is also implemented in the backend-specific namespace. This means that after set_backend is called, the functions in ivy.__dict__ will be the versions that are specific to the chosen backend.

Here's a simplified version of the set_backend function for illustration:

```
# ivy/backend_handler.py
def set_backend(backend: str):

    # un-modified ivy.__dict__
    global ivy_original_dict
    if not backend_stack:
        ivy_original_dict = ivy.__dict__.copy()

    # add the input backend to global stack
    backend_stack.append(backend)

    # iterate through original ivy.__dict__
    for k, v in ivy_original_dict.items():

        # if method doesn't exist in the backend
        if k not in backend.__dict__:
            # add the original ivy method to backend
            backend.__dict__[k] = v
        # update global ivy.__dict__ with this method
        ivy.__dict__[k] = backend.__dict__[k]

    # maybe log to terminal
    if verbosity.level > 0:
        verbosity.cprint(
            'Backend stack: {}'.format(backend_stack))
```

In this code, ivy_original_dict is a copy of the original ivy.__dict__ before any backend was set. The function then adds the chosen backend to the backend_stack, which is a list that keeps track of the backends that have been set.

Next, the function iterates through each item in ivy_original_dict. If a method from ivy_original_dict does not exist in the backend's dictionary (backend.__dict__), it adds the method to backend.__dict__. Then, it updates ivy.__dict__ with the method from backend.__dict__. This effectively overwrites the original Ivy method with the backend-specific version.

Finally, if the verbosity level is greater than 0, it prints the current backend_stack to the terminal.

This process ensures that when you call an Ivy function, the version of the function that gets executed is the one that's specific to the currently set backend.

You can find more details about this in the [Ivy documentation].

#last_update

The functions that are created within a specific backend, like ivy.functional.backends.torch, are only a part of the complete Ivy API. This is because many of the more advanced functions are constructed by combining simpler Ivy functions. This means that these more complex functions don't have to be created separately for each individual backend framework. A clear example of this is the function ivy.lstm_update(), which is illustrated below.

Can you clarify this for me:

The functions implemented by the backend-specific backend such as ivy.functional.backends.torch only constitute a subset of the full Ivy API. This is because many higher-level functions are written as a composition of lower-level Ivy functions. These functions therefore do not need to be written independently for each backend framework. A good example is ivy.lstm_update(), as shown:

More explanation about it:

In Ivy, there are different types of functions: primary, compositional, and mixed. Primary functions are the lowest level building blocks in Ivy and are generally implemented as light wrapping around an existing function in the backend framework, which serves a near-identical purpose. These primary functions are implemented by the backend-specific backend such as ivy.functional.backends.torch and they only constitute a subset of the full Ivy API (Glossary: source).

On the other hand, many higher-level functions in Ivy are written as a composition of these lower-level primary functions. These compositional functions do not have backend-specific implementations. Instead, they are implemented once and can be used across all supported backends. This is because they are written in terms of other Ivy functions, which themselves handle the backend-specific details (Function Types: source).

A good example of this is the ivy.lstm_update() function. This function is implemented as a composition of lower-level Ivy functions, and therefore does not need to be written independently for each backend framework.

Here is the code for ivy.lstm_update():

```
# ivy/functional/ivy/layers.py
@to_native_arrays_and_back
@handle_nestable
def lstm_update(
    x: Union[ivy.Array, ivy.NativeArray],
    init_h: Union[ivy.Array, ivy.NativeArray],
    init_c: Union[ivy.Array, ivy.NativeArray],
    kernel: Union[ivy.Array, ivy.NativeArray],
    recurrent_kernel: Union[ivy.Array, ivy.NativeArray],
    bias: Optional[Union[ivy.Array, ivy.NativeArray]] = None,
    recurrent_bias: Optional[Union[ivy.Array, ivy.NativeArray]] = None,
) -> Tuple[ivy.Array, ivy.Array]:
    # function implementation...
```

The lstm_update function in Ivy is used to perform a long-short term memory update by unrolling the time dimension of the input array. Here are the parameters of the function:

- **x**: This is the input tensor of the LSTM layer. It is expected to have the shape *[batch_shape, t, in]*. It can be an Ivy Array or a Native Array.
- **init_h**: This is the initial state tensor for the cell output. It is expected to have the shape *[batch_shape, out]*. It can be an Ivy Array or a Native Array.
- **init_c**: This is the initial state tensor for the cell hidden state. It is expected to have the shape *[batch_shape, out]*. It can be an Ivy Array or a Native Array.
- **kernel**: This is the weights for the cell kernel. It is expected to have the shape *[in, 4 x out]*. It can be an Ivy Array or a Native Array.
- **recurrent_kernel**: This is the weights for the cell recurrent kernel. It is expected to have the shape *[out, 4 x out]*. It can be an Ivy Array or a Native Array.
- **bias**: This is the bias for the cell kernel. It is expected to have the shape *[4 x out]*. It can be an Ivy Array or a Native Array. This is an optional parameter and its default value is None.
- **recurrent_bias**: This is the bias for the cell recurrent kernel. It is expected to have the shape *[4 x out]*. It can be an Ivy Array or a Native Array. This is an optional parameter and its default value is None.

The function returns a tuple of Ivy Arrays. The first element of the tuple is the hidden state for all timesteps with shape *[batch_shape,t,out]* and the second element is the cell state for the last timestep with shape *[batch_shape,out]*.

You can find more details about this function in the Ivy documentation.

This design allows Ivy to provide a user-friendly API that abstracts away many of the low-level details of building and training ML models, allowing users to focus on designing and experimenting with their models.

A. 4. Ivy Compiler ✅

The compiler accepts various inputs, including Ivy functions, backend functions, and compositions. It then generates a computation graph using only the backend's functional API. The dependency graph illustrating this process is depicted in the following link: Image Link.

B. Ivy as a Transpiler:

B. 1. Front-end functional APIs 🚧

With Ivy's transpiler, you can easily integrate code from different frameworks, even if they are different versions of the same framework. This integration is achieved with just a single line of code. Behind the scenes, Ivy works by tracing a computational graph and utilizing both frontends and backends to establish connections between different frameworks.

In this manner, Ivy opens up access to a wide range of machine learning projects, regardless of the framework you intend to use for research, development, or system deployment. For a comprehensive understanding, you can explore the full API reference in the documentation. However, the following functions are likely to be of particular interest:

- **ivy.compile():** This function compiles a given function into an optimized, complete graph, eliminating any unnecessary wrapping or redundant code.
- **ivy.transpile():** With this function, you can convert code that's specific to one framework into code that is compatible with a different framework.
- **ivy.unify():** This function facilitates the conversion of framework-specific code into the Ivy framework.

These functions offer both eager and lazy usage options. When you provide the required arguments for function tracing, the compilation or transpilation process occurs immediately (eager usage). On the other hand, if you don't provide these arguments, the compilation or transpilation takes place only when the resulting function is first called (lazy usage).

C. Ivy as a Framework:

C. 1. Ivy Stateful API ✅

In this section, we delve into how Ivy's stateful API extends the functional API and the ivy.Container class to introduce additional user-friendly classes like optimizers, network layers, and custom trainable modules. These components are designed to expedite the process of launching your machine learning projects.

Without any delay, let's proceed to explore the features and benefits that the stateful API brings to the table!

#Modules

One of the most valuable classes in Ivy's stateful framework is the ivy.Module. It proves particularly useful for crafting custom layers that can be trained, or even for constructing complete networks. To define trainable variables manually, you need to specify them within the _create_variables() method. For instance, you can create a linear layer by inheriting from ivy.Module, as demonstrated below:

here's the provided code snippet explaining how to create a custom linear layer by inheriting from the ivy.Module class:

```python
class Linear(ivy.Module):

    def __init__(self, input_channels, output_channels,
                 with_bias=True, dev=None, v=None):
        self._input_channels = input_channels
        self._output_channels = output_channels
        self._w_shape = (output_channels, input_channels)
        self._b_shape = (output_channels,)
        self._with_bias = with_bias
        ivy.Module.__init__(self, dev, v)

    def _create_variables(self, dev):
        v = {'w': ivy.random_uniform(
                shape=self._w_shape, dev=dev)}
        if self._with_bias:
            v = dict(**v, b=ivy.random_uniform(
                        shape=self._b_shape, dev=dev))
        return v

    def _forward(self, inputs):
        return ivy.linear(
          inputs, self.v.w,
          self.v.b if self._with_bias else None)
```

This code defines a custom Linear class that extends ivy.Module. It allows you to create a linear layer with customizable parameters such as input and output channels, bias inclusion, device selection, and initial values (v). The _create_variables method defines the trainable variables for the layer, while the _forward method performs the linear transformation using Ivy's linear function. This example showcases the versatility of Ivy's stateful API in creating custom neural network layers.

Please note that, to keep things simple, the provided implementation differs slightly from the built-in ivy.Linear in a few aspects, as we'll elaborate in the subsequent "Initializer" section (see Ivy's docs).

Every instance of ivy.Module contains an attribute called v, which stands for "variables." This attribute holds all the trainable variables within the module, organized within an ivy.Container. For the example we discussed earlier, the hierarchical arrangement of these variables mirrors what's defined in the _create_variables() method.

C. 2. Ivy Container ✅

The ivy.Container class can be likened to an enhanced version of a dictionary, equipped with numerous beneficial functionalities. It forms the foundation for many advanced operations within Ivy.

Now, let's delve into some of the key features that the ivy.Container offers!

#Construction

A Container can be created using various methods. All the approaches listed below lead to the creation of identical ivy.Container instances.

```
import ivy

# Dictionary
dct = {'a': ivy.array([0.]),
       'b': {'c': ivy.array([1.]),
             'd': ivy.array([2.])}}

# Via dict
cnt = ivy.Container(dct)

# Via keyword
cnt = ivy.Container(a=ivy.array([0.]),
                    b=ivy.Container(c=ivy.array([1.]),
                                    d=ivy.array([2.])))

# Combos
cnt = ivy.Container(a=ivy.array([0.]),
                    b={'c': ivy.array([1.]),
                       'd': ivy.array([2.])})

cnt = ivy.Container({'a': ivy.array([0.]),
                     'b': ivy.Container(c=ivy.array([1.]),
                                        d=ivy.array([2.]))})
```

These examples showcase different approaches to create an ivy.Container. The container can store arrays and nested dictionaries, providing a flexible structure for managing data in a hierarchical manner.

Please note that there are more ivy Container features such as **#Representation**, **#Recursive Methods** and so on that you can find here.

C. 3. Ivy Array ✅

In this section, we provide an explanation of the ivy.Array class, which serves as the representation for all arrays within Ivy. Every method in Ivy, when it returns arrays, actually returns instances of the ivy.Array class. This class is fundamental for handling arrays and serves as the common structure for all arrays generated and processed within Ivy.

#The Array Class

Please jump right in and take a look at the very important structure of the `ivy.Array` constructor.

For more detailed information for this section, please see the Ivy documentation.

2. Ivy's Deep Learning

Exploring the sections mentioned below will take you deep into the details of the framework 🌱. This should help you grasp a clearer picture of what's happening behind the scenes 🎬.

Please note that there are more than five frameworks. Clicking on one of the links below will take you to the full documentation.

(a) Navigating the Code 🧭

A quick tour through the codebase that covers explanation on: 1. Navigating the Code, 2. Submodule Desing, 3. Ivy API, 4. Backend API, 5. Submodule Helper Functions, and 6. Version Unpinning.

(b) Function Types 🧮

Primary, compositional, mixed, and nestable functions. This section helps to understand Ivy's different types of functions such as,

1. Primary Function, 2. Compositional Functions, Mixed Functions, Partial Mixed Functions, Standalone Functions, and Nestable Functions.

(c) Superset Behaviour ⊃

Ivy goes for the superset when unifying the backend functions. This section helps contributors with questions such as "which backend implementation should Ivy most closely follow? It covers:

1. Extending the Standard, 2. What is the superset? 3. A Non-Duplicate Superset,

4. What is not the Superset? 5. Bala Balancing Generalization with Efficiency,

6. More Examples, 7. Maximizing Usage of Native Functionality.

(d) Backend Setting ⚙

How the backend is set, and what this means for each function type. This part explains how the backend framework can either be set by calling 'ivy.set_backend(backend_name)' or it can be inferred from the arguments. This part helps regarding both:

1. Dynamic Backend Setting and 2. Backend and Frontend Version Support.

(e) Arrays 🔢

Different types of arrays, and how they're handled. Note that, in Ivy there are two types: 'ivy.NativeArray' and 'ivy.Array'. The Arrays section covers:

1. Native Array, 2. Ivy Array, 3. Array Handling, 4. Integrating Custom Classes with Ivy.

(f) Containers 🚚

You can find additional sub-sections such as Containers, Data-Types, etc. here.

2.1. Deep Dive's Function Examples

How to set up and use Ivy model?

Sure, I can provide you with a Python script based on the information extracts provided. Here's an example of how to set up and use an Ivy model:

```
# Setting up
# Clone the Ivy models repository and install it
!git clone https://github.com/unifyai/models
%cd models
!pip install .

# Getting started
import ivy
from ivy_models import alexnet
```

```
# Set the backend to PyTorch
ivy.set_backend("torch")

# Load the pretrained AlexNet model
model = alexnet()
```

This script clones the Ivy models repository, installs it, and then uses Ivy to load a pretrained AlexNet model. The backend is set to PyTorch, but you can change this to any other supported backend (JAX, TensorFlow, or NumPy) by changing the argument to `ivy.set_backend`.

Please note that this script should be run in an environment where Ivy and the necessary backend are installed. You can install Ivy using pip (`pip install ivy`) or Docker, as described in the **Ivy documentation**.

Also, please note that this script uses the `!` and `%` prefixes for shell commands in a Jupyter notebook. If you're running this script in a regular Python environment, you should remove these prefixes and run the shell commands (`git clone`, `cd`, and `pip install`) separately in your shell.

3. Ivy as a Framework

On the Ivy as a transpiler (One of Ivy's important functionalities) page, we detailed the function of the backend-specific frontends within Ivy. These frontends play a crucial role in facilitating automatic code conversion between various machine learning frameworks. This specific section is identified as part (b) in the accompanying image shown here Design — Ivy Documentation (unify.ai)

Up to this point, taking into account sections (a) and (b) -- as we viewed it from the above link -- our focus has primarily been on viewing Ivy as a comprehensive framework equipped with code conversion capabilities. Ivy takes these foundational elements and further develops them to establish a complete machine learning framework. This includes incorporating stateful classes, optimizers, and user-friendly utilities, all designed to streamline the process of running machine learning experiments using just a minimal amount of code.

In this context, our focus is directed towards the ivy.Container class, the ivy.Array class, and the stateful API. These particular components are denoted as part (c) within the image found at the following link: Image Link.

Please feel free to refer to "**C. Ivy as a Framework**" as well for this part.

4. Ivy Frontends

#Introduction

In addition to Ivy's functional API and backend functional APIs, there's an additional collection of framework-specific frontend functional APIs. These APIs are crucial in the process of code transpilation, which is explained more upon here.

#The Frontend Basics

When utilizing functions and methods from Ivy Frontends, alongside importing **ivy** itself using **import ivy**, it's important to also import the relevant Frontend module. For instance, if you're working with Ivy's TensorFlow frontend, you would import it as follows:

```python
import ivy.functional.frontends.tensorflow as tf_frontend
```

This ensures that you have access to the specific frontend functionalities that are relevant to the framework you're using.

When conducting tests on frontend functions, there are instances when we can directly call the function from the root frontend namespace. For instance, we might use **tensorflow.tan** instead of **tensorflow.math.tan()**. In this specific scenario, both approaches are acceptable and are essentially interchangeable aliases for each other.

However, there are cases where an additional namespace path becomes necessary. Let's consider JAX as an example. In JAX, there are functions like **jax.numpy.abs()** and **jax.lax.abs(),** but **jax.abs()** does not exist. In our JAX frontend, if we include both of these functions in the root namespace, it would prevent us from directly calling **jax.abs()** within our frontend.

This situation could lead to **jax.numpy.abs()** or **jax.lax.abs()** overwriting each other in an unpredictable manner. However, it's important to note that neither of these functions should be included in the root namespace, as they don't exist in the native JAX framework.

If, by mistake, you test a function using **fn_tree="<func_name>"** instead of **fn_tree="<lax|numpy>.<func_name>"**, you will encounter an error. This error occurs because you're attempting to test the wrong frontend function.

To prevent such conflicts, it's important to take the following steps:

1. All frontend tests must use the complete namespace path when calling the frontend function. For TensorFlow, this involves using **fn_tree="math.tan"** instead of **fn_tree="tan"** in the frontend test.
2. Carefully inspect the **__init__.py** file within all frontends. Ensure that you're not introducing aliases that shouldn't exist, like the example of **jax.abs()** mentioned earlier.
3. Prior to merging any frontend pull requests, make sure that all tests are passing. The only exception to this rule is if a test fails due to a bug in the Ivy functional API, which doesn't need to be resolved as part of the frontend task.

In these examples, there will be implicit discussions about the placement of frontend functions. For a clear guide on how to position or place a frontend function, refer to the sub-section within the Frontend APIs open task, fully explained here.

NOTE: Type hints, docstrings and examples are not required when working on frontend functions.

What is the meaning of “namespace pat” in this context?

In this context, "namespace path" refers to the full hierarchical naming structure that uniquely identifies a function within a particular programming library or framework. It includes all the nested levels of modules or sub-modules that lead to the specific function. This naming structure helps avoid ambiguity and ensures that the correct function is being accessed, especially when multiple functions might have similar names but exist in different parts of the library.

For example, in TensorFlow, the function **math.tan** has a namespace path of **tensorflow.math.tan.** Using the complete namespace path ensures that there's no confusion about which **tan** function is being referred to, even if there are other functions named **tan** in different parts of the library.

For comprehensive Ivy documentation on Ivy Frontends, please click here to access additional details about **#Front Array, #Writing Frontend Functions** (for Jax, Numpy, …), and more.

5. Contributing

Ivy wholeheartedly encourages contributions and active engagement from individuals across the community, regardless of their level of expertise.

Whether you're an experienced developer 🥷 or just starting out 🤷, everyone is invited to participate by raising issues and submitting pull requests. Your input is valued and welcomed in advancing the project.

Ivy Decorators

Ivy provides test decorators for frontend tests to make it easier and more maintainable, currently there are two:

1. **@handle_frontend_test()**: A decorator which is used to test frontend functions, for example np.zeros() and tensorflow.tan().

2. **@handle_frontend_method()**: A decorator which is used to test frontend methods and special methods, for example torch.Tensor.add() and numpy.ndarray.__add__().

What is the difference between a method and a function in this context in the above?

In this context, the distinction between a "method" and a "function" is as follows:

A **function** is a standalone piece of code that can be called and executed. In this case, Ivy provides test decorators like **@handle_frontend_test()** to test frontend functions like **np.zeros()** and **tensorflow.tan()**. These are independent operations that take input and return output.

On the other hand, a **method** is a function that is associated with an object and can operate on its data. In this context, the decorator **@handle_frontend_method()** is used to test frontend methods and special methods. For instance, **torch.Tensor.add()** and **numpy.ndarray.__add__()** are examples of methods. These methods are associated with specific objects (tensor and ndarray) and often perform operations on the data contained within those objects.

So, in short, functions are standalone operations, while methods are functions tied to specific objects and operate on their data.

Important Helper Functions

The main simple difference between them is that they are **helper functions** that serve different purposes:

1. **helpers.test_frontend_function()**: This helper function simplifies testing Ivy Frontends by handling complex tasks. It compares the result of a frontend function with the function in the associated framework to ensure accurate results.
2. **helpers.get_dtypes()**: This helper function provides data types. It can return a list of all data types or a single data type. It's recommended to use this function for sampling data types.
3. **helpers.dtype_and_values()**: This convenience function generates arrays with various dimensions and their corresponding data types. It returns both a list of data types and corresponding NumPy arrays.
4. **helpers.get_shape()**: This convenience function generates array shapes as tuples, making it easier to work with array dimensions.
5. **np_frontend_helpers.where()**: This function is a generation strategy specifically tailored for generating values for NumPy's optional **where** argument. It assists in creating values that can be used with the **where** argument in NumPy functions.
6. **np_frontend_helpers.test_frontend_function()**: This function is similar to **helpers.test_frontend_function()**, but it's designed to handle NumPy's optional **where** argument. It behaves identically to the general helper function, but it's specialized to accommodate the unique behavior of the **where** argument in NumPy functions.

In summary, these helper functions are designed to simplify tasks related to testing, data type handling, and array manipulation.

Useful Note

Can you explain to me why these functions are different?

The test_frontend_function() argument fn_tree refers to the frontend function's reference in its native namespace not just the function name. For example lax.tan() is needed for some functions in Jax, nn.functional.relu() is needed for some functions in PyTorch etc.

Certainly, the distinction between these functions lies in their treatment of the **fn_tree** argument and their specific use cases within different frameworks:

1. **test_frontend_function()**: This function requires the **fn_tree** argument, which refers to the reference of the frontend function within its native namespace. This means you need to provide the whole function reference, including its namespace, not just the function name. For instance, when using **lax.tan()** in Jax or **nn.functional.relu()** in PyTorch, you would pass these complete function references as **fn_tree**. This function is designed to work with various frameworks and their specific function references.

2. **np_frontend_helpers.test_frontend_function()**: This function, specific to NumPy, is similar to the general **test_frontend_function()**. However, it takes into account NumPy's optional **where** argument. It also requires the **fn_tree** argument in the same way as the general version, referencing the frontend function within its namespace. The main difference here is its specialization for NumPy and its handling of the **where** argument.

In summary, both functions have a similar purpose of testing frontend functions, but they are designed to accommodate different frameworks (general use and NumPy-specific) and require the complete function reference (**fn_tree**) for accurate testing within their respective namespaces.

An Example on Testing array_api_testing **Module** of **Ivy's Functions** for **Contributing**

To get into array_api_testing in Ivy, you need to follow these steps ():

1. Fork and clone the Ivy repository to your local machine. You can do this by forking the repository from the **repository page** and then cloning it to your local machine. Here's how you can clone it:

```
git clone --recurse-submodules git@github.com:YOUR_USERNAME/ivy.git
```

or

```
git clone --recurse-submodules https://github.com/YOUR_USERNAME/ivy.git
```

or

```
gh repo clone YOUR_USERNAME/ivy your_folder -- --recurse-submodules
```

Then, navigate to your cloned Ivy folder and add the Ivy original repository as upstream to easily sync with the latest changes:

```
git remote add upstream https://github.com/unifyai/ivy.git
```

(Source: **Setting Up**)

2. Ivy uses the Array API test suite to ensure that all backend libraries behave according to the Array API Standard. The test suite is included in the Ivy repository as a submodule in the folder test_array_api. You can update the submodule with the following commands:

```
# to initialise local config file and fetch + checkout submodule (not needed everytime)
git submodule update --init --recursive

# pulls changes from upstream remote repo and merges them
git submodule update --recursive --remote --merge
```

(Source: **Array API Tests**)

3. To run the tests, you can use either the terminal or your IDE. If you're using the terminal, you can run all array-api tests in a given file for a certain backend using the bash file test_array_api.sh:

```
# /ivy
/bin/bash -e ./run_tests_CLI/test_array_api.sh jax test_linalg
```

You can replace jax with any of the supported frameworks - tensorflow, numpy, torch - and replace test_linalg with the individual test function categories in ivy/ivy_tests/array_api_testing/test_array_api/array_api_tests, e.g. test_set_functions, test_signatures etc.

To run a specific test, you should set the backend explicitly in the _array_module.py file, which you can find in the array_api_tests submodule. At the beginning of the file, you will see the following line of code array_module = None. You need to comment out that line and add the following:

```
import ivy as array_module
array_module.set_backend("jax") # or numpy, tensorflow, torch
```

Then, you can run the following commands via terminal:

```
# run all tests in a file
pytest -vv
ivy_tests/array_api_testing/test_array_api/array_api_tests/test_manipulation_functions.py

# run a single test
pytest -vv
ivy_tests/array_api_testing/test_array_api/array_api_tests/test_manipulation_functions.py -k
"test_concat"
```

(Source: **Array API Tests > Running the Tests > Using Terminal**)

If you have any questions, you can reach out on **discord**. (Source: **Ivy Array > Instance Methods Part 2**)

Answer based on the following sources:
github.com/unifyai
unify.ai/docs
unify.ai/docs
unify.ai/docs
discord.gg/sXyFF8tDtm
unify.ai/docs

6. Conclusion

Design: This section offered an introductory insight into Ivy's architecture and foundational elements. It was ideal for grasping the framework's structure and functionality. It served dual purposes: facilitating automatic code conversions among frameworks and operating as a versatile ML framework.

Deep Dive: Intended for those interested in delving into Ivy's inner workings, this section covered diverse aspects, including code navigation, function types, backend settings, and more. It was a comprehensive resource for understanding Ivy's mechanics.

Ivy as a Framework: Explored Ivy's role as a comprehensive ML framework, this section delved into the ivy.Container and ivy.Array classes, along with the stateful API. It illuminated how Ivy could be harnessed for ML projects.

Ivy Frontends: Detailed the significance of backend-specific frontends, this section explained their role in enabling seamless code conversions across diverse ML frameworks. You were encouraged to tap into the Ivy community for queries or clarifications.

Contributing: If you were eager to contribute to Ivy, this section served as a guide to kick-start your involvement. It outlined setting up your environment, engaging with issues and pull requests, and building documentation

In essence, this guide helps you grasp Ivy's functioning and its potential for building and training ML models. Should you have questions, the Ivy community is there to assist you.