The Kybra Book

This book is intended to be an in-depth guide to canister development in Python on the Internet Computer (IC).

The first 19 chapters are an introductory guide into canister development with Kybra. These chapters build on each other concept by concept, introducing the fundamentals required to create and deploy canisters to the IC.

Chapter 20 is an in-depth reference of the APIs available to Kybra canisters.

Our intention is for new developers to use this book as a tutorial or course, starting at chapter 1 and working through chapter 19, using chapter 20 as a reference.

There will also be a companion video series on YouTube. Each chapter here will begin with the video companion as soon as it is available.

You should expect this book and its companion video series to continue to grow and change over time, as its authors and the IC grow and change.

The Kybra Book is subject to the following license:

Kybra (Beta)

Kybra is a Python Canister Development Kit (CDK) for the Internet Computer (IC). In other words, it's a Python runtime for building applications (canisters) on the IC.

- GitHub repo
- PyPI package
- Discord channel

Disclaimer

Kybra may have unknown security vulnerabilities due to the following:

- Kybra does not yet have many live, successful, continuously operating applications deployed to the IC
- Kybra does not yet have extensive automated property tests
- Kybra does not yet have multiple independent security reviews/audits
- Kybra uses a new Python interpreter that is less mature than CPython

Roadmap

We hope to get to a production-ready 1.0 in 2024. The following are the major blockers to 1.0:

- CPython integration for performance, security, and stability
- Broad PyPI package support (C API/extensions)
- Extensive automated property testing
- Multiple independent security reviews/audits

Demergent Labs

Kybra is currently developed by Demergent Labs, a for-profit company with a grant from DFINITY.

Demergent Labs' vision is to accelerate the adoption of Web3, the Internet Computer, and sustainable open source.

Benefits and drawbacks

Kybra and the IC provide unique benefits and drawbacks, and both are not currently suitable for all application use-cases.

The following information will help you to determine when Kybra and the IC might be beneficial for your use-case.

Benefits

Kybra intends to be a full Python environment for the IC (a decentralized cloud platform), with support for all of the Python language and as many relevant libraries and APIs as possible.

One of the core benefits of Kybra is that it allows Python developers to bring their skills to the IC.

As for the IC, we believe its main benefits can be broken down into the following categories:

- Ownership
- Security
- Developer Experience

Most of these benefits stem from the decentralized nature of the IC, though the IC is best thought of as a progressively decentralizing cloud platform. As opposed to traditional cloud platforms, its goal is to be owned and controlled by many independent entities.

Ownership

- Full-stack group ownership
- Autonomous ownership
- Permanent APIs
- Credible neutrality
- Reduced platform risk

Full-stack group ownership

The IC allows you to build applications that are controlled directly and only (with some caveats) by a group of people. This is in opposition to most cloud applications written today, which must be under the control of a very limited number of people and often a single legal entity that answers directly to a cloud provider, which itself is a single legal entity.

In the blockchain world, group-owned applications are known as DAOs. As opposed to DAOs built on most blockchains, the IC allows full-stack applications to be controlled by groups. This means that the group fully controls the running instances of the frontend and the backend code.

Autonomous ownership

In addition to allowing applications to be owned by groups of people, the IC also allows applications to be owned by no one. This essentially creates autonomous applications or everlasting processes that execute indefinitely. The IC will allow such an application to run until it depletes its balance of cycles, or until the NNS votes to shut it down.

Permanent APIs

Because most web APIs are owned and operated by individual entities, their fate is tied to that of their owners. If their owners go out of business, then those APIs may cease to exist. If their owners decide that they do not like or agree with certain users, they may restrict their access. In the end, they may decide to shut down or restrict access for arbitrary reasons.

Because the IC allows for group and autonomous ownership of cloud software, the IC is able to produce potentially permanent web APIs. A decentralized group of independent entities will find it difficult to censor API consumers or shut down an API. An autonomous API would take those difficulties to the extreme, as it would continue operating as long as consumers were willing to pay for it.

Credible neutrality

Group and autonomous ownership makes it possible to build neutral cloud software on the IC. This type of software would allow independent parties to coordinate with reduced trust in each other or a single third-party coordinator.

This removes the risk of the third-party coordinator acting in its own self-interest against the interests of the coordinating participants. The coordinating participants would also find it difficult to implement changes that would benefit themselves to the detriment of other participants.

Examples could include mobile app stores, ecommerce marketplaces, and podcast directories.

Reduced platform risk

Because the IC is not owned or controlled by any one entity or individual, the risk of being deplatformed is reduced. This is in opposition to most cloud platforms, where the cloud provider itself generally has the power to arbitrarily remove users from its platform.

While deplatforming can still occur on the IC, the only endogenous means of forcefully taking down an application is through an NNS vote.

Security

- Built-in replication
- Built-in authentication
- Built-in firewall/port management
- Built-in sandboxing
- Threshold protocols
- Verifiable source code
- Blockchain integration

Built-in replication

Replication has many benefits that stem from reducing various central points of failure.

The IC is at its core a Byzantine Fault Tolerant replicated compute environment. Applications are deployed to subnets which are composed of nodes running replicas. Each replica is an independent replicated state machine that executes an application's state transitions (usually initiated with HTTP requests) and persists the results.

This replication provides a high level of security out-of-the-box. It is also the foundation of a number of protocols that provide threshold cryptographic operations to IC applications.

Built-in authentication

IC client tooling makes it easy to sign and send messages to the IC, and Internet Identity provides a novel approach to self-custody of private keys. The IC automatically authenticates messages with the public key of the signer, and provides a compact representation of that public key, called a principal, to the application. The principal can be used for authorization purposes. This removes many authentication concerns from the developer.

Built-in firewall/port management

The concept of ports and various other low-level network infrastructure on the IC is abstracted away from the developer. This can greatly reduce application complexity thus minimizing the chance of introducing vulnerabilities through incorrect configurations. Canisters expose endpoints through various methods, usually query or update methods. Because authentication is also built-in, much of the remaining vulnerability surface area is minimized to implementing correct authorization rules in the canister method endpoints.

Built-in sandboxing

Canisters have at least two layers of sandboxing to protect colocated canisters from each other. All canisters are at their core Wasm modules and thus inherit the built-in Wasm sandbox. In case there is any bug in the underlying implementation of the Wasm execution environment (or a vulnerability in the imported host functionality), there is also an OS-level sandbox. Developers need not do anything to take advantage of these sandboxes.

Threshold protocols

The IC provides a number of threshold protocols that allow groups of independent nodes to perform cryptographic operations. These protocols remove central points of failure while providing familiar and useful cryptographic operations to developers. Included are ECDSA, BLS, VRF-like, and in the future threshold key derivation.

Verifiable source code

IC applications (canisters) are compiled into Wasm and deployed to the IC as Wasm modules. The IC hashes each canister's Wasm binary and stores it for public retrieval. The Wasm binary hash can be retrieved and compared with the hash of an independently compiled Wasm binary derived from available source code. If the hashes match, then one can know with a high degree of certainty that the application is executing the Wasm binary that was compiled from that source code.

For the time being, Kybra source code is not verifiable for reasons explained in the caveats section.

Blockchain integration

When compared with web APIs built for the same purpose, the IC provides a high degree of security when integrating with various other blockchains. It has a direct client integration with Bitcoin, allowing applications to query its state with BFT guarantees. A similar integration is coming for Ethereum.

In addition to these blockchain client integrations, a threshold ECDSA protocol (tECDSA) allows the IC to create keys and sign transactions on various ECDSA chains. These chains include Bitcoin and Ethereum, and in the future the protocol may be extended to allow interaction with various EdDSA chains. These direct integrations combined with tECDSA provide a much more secure way to provide blockchain functionality to end users than creating and storing their private keys on traditional cloud infrastructure.

Developer experience

- Built-in devops
- Orthogonal persistence

Built-in devops

The IC provides many devops benefits automatically. Though currently limited in its scalability, the protocol attempts to remove the need for developers to concern themselves with concepts such as autoscaling, load balancing, uptime, sandboxing, and firewalls/port management.

Correctly constructed canisters have a simple deploy process and automatically inherit these devops capabilities up unto the current scaling limits of the IC. DFINITY engineers are constantly working to remove scalability bottlenecks.

Orthogonal persistence

The IC automatically persists its heap. This creates an extremely convenient way for developers to store application state, by simply writing into global variables in their programming language of choice. This is a great way to get started.

If a canister upgrades its code, swapping out its Wasm binary, then the heap must be cleared. To overcome this limitation, there is a special area of memory called stable memory that persists across these canister upgrades. Special stable data structures provide a familiar API that allows writing into stable memory directly.

All of this together provides the foundation for a very simple persistence experience for the developer. The persistence tools now available and coming to the IC may be simpler than their equivalents on traditional cloud infrastructure.

Drawbacks

It's important to note that both Kybra and the IC are early-stage projects. The IC officially launched in May of 2021, and Kybra reached beta in December of 2022.

Kybra

Some of Kybra's main drawbacks can be summarized as follows:

- Beta
- Security risks
- High cycle usage
- Missing APIs

Beta

Kybra reached beta in December of 2022. It's an immature project that may have unforeseen bugs and other issues. We're working constantly to improve it. We hope to get to a production-ready 1.0 in 2024. The following are the major blockers to 1.0:

- CPython integration for performance, security, and stability
- Broad PyPI package support (C API/extensions)
- Extensive automated property testing
- Multiple independent security reviews/audits

Security risks

As discussed earlier, these are some things to keep in mind:

- Kybra does not yet have many live, successful, continuously operating applications deployed to the IC
- Kybra does not yet have extensive automated property tests
- Kybra does not yet have multiple independent security reviews/audits
- Kybra uses a new Python interpreter that is less mature than CPython

High cycle usage

We haven't done extensive benchmarking yet, but based on some preliminary evidence Kybra is likely much more performant than Azle. We have done some preliminary benchmarking for Azle, and based on that our rough heuristic is that Azle will cost 2-4x more cycles than the equivalent project in Motoko or Rust. The performance of your application depends on many factors, and this should just be a rough estimate.

There is evidence to suggest that a 7-20x improvement in performance is possible in our underlying Python interpreter. We also plan to migrate to CPython which would improve performance.

Missing APIs

Kybra is limited to what the IC is capable of (i.e. no sockets, no threads, etc) and does not yet have C extension support. Our goal is to support as many libraries and APIs as possible over time.

IC

Some of the IC's main drawbacks can be summarized as follows:

- Early
- High latencies
- Limited and expensive compute resources
- Limited scalability
- Lack of privacy
- NNS risk

Early

The IC launched officially in May of 2021. As a relatively new project with an extremely ambitious vision, you can expect a small community, immature tooling, and an unproven track record. Much has been delivered, but many promises are yet to be fulfilled.

High latencies

Any requests that change state on the IC must go through consensus, thus you can expect latencies of a few seconds for these types of requests. When canisters need to communicate with each other across subnets or under heavy load, these latencies can be even longer. Under these circumstances, in the worst case latencies will build up linearly. For example, if canister A calls canister B calls canister C, and these canisters are all on different subnets or under heavy load, then you might need to multiply the latency by the total number of calls.

Limited and expensive compute resources

CPU usage, data storage, and network usage may be more expensive than the equivalent usage on traditional cloud platforms. Combining these costs with the high latencies explained above, it becomes readily apparent that the IC is currently not built for high-performance computing.

Limited scalability

The IC might not be able to scale to the needs of your application. It is constantly seeking to improve scalability bottlenecks, but it will probably not be able to onboard millions of users to your traditional web application.

Lack of privacy

You should assume that all of your application data (unless it is end-to-end encrypted) is accessible to multiple third-parties with no direct relationship and limited commitment to you. Currently all canister state sits unencrypted on node operator's machines. Application-layer access controls for data are possible, but motivated node operators will have an easy time getting access to your data.

NNS risk

The NNS has the ability to uninstall any canister and can generally change anything about the IC. As of the time of this writing, DFINITY effectively controls much of the NNS through its follower relationships. The NNS must mature and decentralize to provide practical and realistic guarantees to canisters and their users.

Installation

Dependencies

Follow the instructions exactly as stated below to avoid issues.

You should be using a *nix environment (Linux, Mac OS, WSL if using Windows) with bash and have the following installed on your system:

- Python 3.10.7
- dfx 0.14.2
- Python VS Code Extension

Python 3.10.7

It is highly recommended to install Python 3.10.7 using pyenv. To do so, use the pyenv installer as shown below:

```
# install pyenv
curl https://pyenv.run | bash

# install Python 3.10.7
~/.pyenv/bin/pyenv install 3.10.7
```

dfx

Run the following command to install dfx 0.14.2:

```
DFX_VERSION=0.14.2 sh -ci "$(curl -fsSL https://sdk.dfinity.org/install.sh)"
```

If after trying to run `dfx` commands you encounter an error such as `dfx: command not found`, you might need to add `$HOME/bin` to your path. Here's an example of doing this in your `.bashrc`:

```
echo 'export PATH="$PATH:$HOME/bin"' >> "$HOME/.bashrc"
```

Python VS Code Extension

It is highly recommended to use VS Code and to install the Microsoft Python extension to get full type checking support from within the editor:

Extension

```
VS Code -> Preferences -> Extensions -> Search for Python by Microsoft and
install it
```

Set python.analysis.typeCheckingMode

Set the setting `python.analysis.typeCheckingMode` to `strict`:

```
VS Code -> Preferences -> Settings -> Search for
python.analysis.typeCheckingMode and set it to strict
```

Common installation issues

Ubuntu

Error:

```
linker cc not found
```

Resolution:

```
sudo apt install build-essential
```

Error:

```
is cmake not installed?
```

Resolution:

```
sudo apt install cmake
```

Error:

```
ERROR: The Python ssl extension was not compiled. Missing the OpenSSL lib
```

Resolution:

You may have the right version of open ssl but you might be missing libssl-dev

```
sudo apt-get install libssl-dev
```

Hello World

Let's build your first application (canister) with Kybra!

Before embarking please ensure you've followed all of the installation instructions.

We'll build a simple `Hello World` canister that shows the basics of importing Kybra, exposing a query method, exposing an update method, and storing some state in a global variable. We'll then interact with it from the command line and from our web browser.

The project directory and file structure

Assuming you're starting completely from scratch, run these commands to setup your project's directory and file structure:

```
mkdir kybra_hello_world
cd kybra_hello_world

mkdir src

touch src/main.py
touch dfx.json
```

Now create and source a virtual environment:

```
~/.pyenv/versions/3.10.7/bin/python -m venv venv
source venv/bin/activate
```

Now install Kybra:

```
pip install kybra
```

Open up `kybra_hello_world` in your text editor (we recommend VS Code with the Microsoft Python extension).

main.py

Here's the main code of the project, which you should put in the `kybra_hello_world/src/main.py` file of your canister:

```
from kybra import query, update, void

# This is a global variable that is stored on the heap
message: str = ''

# Query calls complete quickly because they do not go through consensus
@query
def get_message() -> str:
    return message

# Update calls take a few seconds to complete
# This is because they persist state changes and go through consensus
@update
def set_message(new_message: str) -> void:
    global message
    message = new_message # This change will be persisted
```

Let's discuss each section of the code.

```
from kybra import query, update, void
```

The code starts off by importing the `query` and `update` decorators from `kybra`, along with the `void` type. The `kybra` module provides most of the Internet Computer (IC) APIs for your canister.

```
# This is a global variable that is stored on the heap
message: str = ''
```

We have created a global variable to store the state of our application. This variable is in scope to all of the functions defined in this module. We have annotated it with a type and set it equal to an empty string.

```
# Query calls complete quickly because they do not go through consensus
@query
def get_message() -> str:
    return message
```

We are exposing a canister query method here. When query methods are called they execute quickly because they do not have to go through consensus. This method simply returns our global `message` variable.

```
# Update calls take a few seconds to complete
# This is because they persist state changes and go through consensus
@update
def set_message(new_message: str) -> void:
    global message
    message = new_message # This change will be persisted
```

We are exposing an update method here. When update methods are called they take a few seconds to complete. This is because they persist changes and go through consensus. A majority of nodes in a subnet must agree on all state changes introduced in calls to update methods. This method accepts a `string` from the caller and will store it in our global `message` variable.

That's it! We've created a very simple getter/setter `Hello World` application. But no `Hello World` project is complete without actually yelling `Hello world` !

To do that, we'll need to setup the rest of our project.

dfx.json

Create the following in `kybra_hello_world/dfx.json` :

```
{
    "canisters": {
        "kybra_hello_world": {
            "type": "custom",
            "build": "python -m kybra kybra_hello_world src/main.py
src/main.did",
            "post_install": ".kybra/kybra_hello_world/post_install.sh",
            "candid": "src/main.did",
            "wasm": ".kybra/kybra_hello_world/kybra_hello_world.wasm",
            "gzip": true
        }
    }
}
```

Local deployment

Let's deploy to our local replica.

First startup the replica:

```
dfx start --background
```

If you want an extra speedy deploy:

```
dfx start --background --artificial-delay 0
```

Then deploy the canister:

```
dfx deploy
```

If you are asked for a password, you'll need to create a new unencrypted dfx identity:

```
dfx identity new test_unencrypted --storage-mode plaintext
dfx identity use test_unencrypted

dfx deploy
```

This is not ideal for security, but Kybra currently does not support encrypted dfx identities.

Interacting with your canister from the command line

Once we've deployed we can ask for our message:

```
dfx canister call kybra_hello_world get_message
```

We should see `("")` representing an empty message.

Now let's yell `Hello World!`:

```
dfx canister call kybra_hello_world set_message '("Hello World!")'
```

Retrieve the message:

```
dfx canister call kybra_hello_world get_message
```

We should see `("Hello World!")`.

Interacting with your canister from the web UI

After deploying your canister, you should see output similar to the following in your terminal:

```
Deployed canisters.
URLs:
  Backend canister via Candid interface:
    kybra_hello_world: http://127.0.0.1:8000/?canisterId=ryjl3-tyaaa-aaaaa-
aaaba-cai&id=rrkah-fqaaa-aaaaa-aaaaq-cai
```

Open up http://127.0.0.1:8000/?canisterId=ryjl3-tyaaa-aaaaa-aaaba-cai&id=rrkah-fqaaa-aaaaa-aaaaq-cai or the equivalent URL from your terminal to access the web UI and interact with your canister.

Deployment

There are two main Internet Computer (IC) environments that you will generally interact with: the local replica and mainnet.

When developing on your local machine, our recommended flow is to start up a local replica in your project's root directoy and then deploy to it for local testing.

Starting the local replica

Open a terminal and navigate to your project's root directory:

```
dfx start
```

Alternatively you can start the local replica as a background process:

```
dfx start --background
```

If you want extra speedy deploys:

```
dfx start --artificial-delay 0
```

or

```
dfx start --background --artificial-delay 0
```

If you want to stop a local replica running in the background:

```
dfx stop
```

If you ever see this error after `dfx stop`:

```
Error: Failed to kill all processes.  Remaining: 627221 626923 627260
```

Then try this:

```
sudo kill -9 627221
sudo kill -9 626923
sudo kill -9 627260
```

If your replica starts behaving strangely, we recommend starting the replica clean, which will clean the `dfx` state of your project:

```
dfx start --clean
```

Deploying to the local replica

To deploy all canisters defined in your `dfx.json`:

```
dfx deploy
```

To deploy an individual canister:

```
dfx deploy canister_name
```

If you are asked for a password, you'll need to create a new unencrypted dfx identity:

```
dfx identity new test_unencrypted --storage-mode plaintext
dfx identity use test_unencrypted

dfx deploy
```

Interacting with your canister

As a developer you can generally interact with your canister in three ways:

- dfx command line
- dfx web UI
- @dfinity/agent

dfx command line

You can see a more complete reference here.

The commands you are likely to use most frequently are:

```
# assume a canister named my_canister

# builds and deploys all canisters specified in dfx.json
dfx deploy

# builds all canisters specified in dfx.json
dfx build

# builds and deploys my_canister
dfx deploy my_canister

# builds my_canister
dfx build my_canister

# removes the Wasm binary and state of my_canister
dfx uninstall-code my_canister

# calls the method_name method on my_canister with a string argument
dfx canister call my_canister method_name '("This is a Candid string
argument")'
```

dfx web UI

After deploying your canister, you should see output similar to the following in your terminal:

```
Deployed canisters.
URLs:
  Backend canister via Candid interface:
    my_canister: http://127.0.0.1:8000/?canisterId=ryjl3-tyaaa-aaaaa-aaaba-
cai&id=rrkah-fqaaa-aaaaa-aaaaq-cai
```

Open up http://127.0.0.1:8000/?canisterId=ryjl3-tyaaa-aaaaa-aaaba-cai&id=rrkah-fqaaa-aaaaa-aaaaq-cai to access the web UI.

@dfinity/agent

@dfinity/agent is the TypeScript/JavaScript client library for interacting with canisters on the IC. If you are building a client web application, this is probably what you'll want to use.

There are other agents for other languages as well:

- Java
- Python
- Rust

Deploying to mainnet

Assuming you are setup with cycles, then you are ready to deploy to mainnet.

To deploy all canisters defined in your dfx.json:

```
dfx deploy --network ic
```

To deploy an individual canister:

```
dfx deploy --network ic canister_name
```

If you are asked for a password, you'll need to create a new unencrypted dfx identity:

```
dfx identity new test_unencrypted --storage-mode plaintext
dfx identity use test_unencrypted

dfx deploy --network ic
```

Examples

Kybra has many example projects showing nearly all Kybra APIs. They can be found in the examples directory of the Kybra GitHub repository.

We'll highlight a few of them here:

- Query
- Update
- Primitive Types
- Stable Structures
- Cycles
- Cross Canister Calls
- Management Canister
- Outgoing HTTP Requests
- Incoming HTTP Requests
- Pre and Post Upgrade
- Timers

Internet Computer Overview

The Internet Computer (IC) is a decentralized cloud platform. Actually, it is better thought of as a progressively decentralizing cloud platform. Its full vision is yet to be fulfilled.

It aims to be owned and operated by many independent entities in many geographies and legal jurisdictions throughout the world. This is in opposition to most traditional cloud platforms today, which are generally owned and operated by one overarching legal entity.

The IC is composed of computer hardware nodes running the IC protocol software. Each running IC protocol software process is known as a replica.

Nodes are assigned into groups known as subnets. Each subnet attempts to maximize its decentralization of nodes according to factors such as data center location and node operator independence.

The subnets vary in size. Generally speaking the larger the size of the subnet the more secure it will be. Subnets currently range in size from 13 to 40 nodes, with most subnets having 13 nodes.

IC applications, known as canisters, are deployed to specific subnets. They are then accessible through Internet Protocol requests such as HTTP. Each subnet replicates all canisters across all of its replicas. A consensus protocol is run by the replicas to ensure Byzantine Fault Tolerance.

View the IC Dashboard to explore all data centers, subnets, node operators, and many other aspects of the IC.

Canisters Overview

Canisters are Internet Computer (IC) applications. They are the encapsulation of your code and state, and are essentially Wasm modules.

State can be stored on the 4 GiB heap or in a larger 64 GiB location called stable memory. You can store state on the heap using your language's native global variables. You can store state in stable memory using low-level APIs or special stable data structures that behave similarly to native language data structures.

State changes must go through a process called consensus. The consensus process ensures that state changes are Byzantine Fault Tolerant. This process takes a few seconds to complete.

Operations on canister state are exposed to users through canister methods. These methods can be invoked through HTTP requests. Query methods allow state to be read and are low-latency. Update methods allow state to be changed and are higher-latency. Update methods take a few seconds to complete because of the consensus process.

Query Methods

TLDR

- Decorate functions with `@query`
- Read-only
- Executed on a single node
- No consensus
- Latency on the order of ~100 milliseconds
- 5 billion Wasm instruction limit
- 4 GiB heap limit
- ~32k queries per second per canister

The most basic way to expose your canister's functionality publicly is through a query method. Here's an example of a simple query method:

```
from kybra import query


@query
def get_string() -> str:
    return "This is a query method!"
```

`get_string` can be called from the outside world through the IC's HTTP API. You'll usually invoke this API from the `dfx command line`, `dfx web UI`, or an agent.

From the `dfx command line` you can call it like this:

```
dfx canister call my_canister get_string
```

Query methods are read-only. They do not persist any state changes. Take a look at the following example:

```
from kybra import query, void

db = {}


@query
def set(key: str, value: str) -> void:
    db[key] = value
```

Calling `set` will perform the operation of setting the `key` item on the `db` dictionary to `value`, but after the call finishes that change will be discarded.

This is because query methods are executed on a single node machine and do not go through consensus. This results in lower latencies, perhaps on the order of 100 milliseconds.

There is a limit to how much computation can be done in a single call to a query method. The current query call limit is 5 billion Wasm instructions. Here's an example of a query method that runs the risk of reaching the limit:

```
from kybra import nat32, query


@query
def pyramid(levels: nat32) -> str:
    levels_array = [0 for _ in range(levels)]
    asterisk_array = [
        ["*" for _ in range(i + 1)] + ["\n"] for i in
range(len(levels_array))
    ]
    flattened_array = [element for subarray in asterisk_array for element in
subarray]

    return "".join(flattened_array)
```

From the `dfx command line` you can call `pyramid` like this:

```
dfx canister call my_canister pyramid '(600)'
```

With an argument of `600`, `pyramid` will fail with an error `...exceeded the instruction limit for single message execution`.

Keep in mind that each query method invocation has up to 4 GiB of heap available.

In terms of query scalability, an individual canister likely has an upper bound of ~36k queries per second.

Update Methods

TLDR

- Annotate functions with `@update`
- Read-write
- Executed on many nodes
- Consensus
- Latency ~2-5 seconds
- 20 billion Wasm instruction limit
- 4 GiB heap limit
- 48 GiB stable memory limit
- ~900 updates per second per canister

Update methods are similar to query methods, but state changes can be persisted. Here's an example of a simple update method:

```
from kybra import nat64, update

counter = 0


@update
def increment() -> nat64:
    global counter
    counter += 1
    return counter
```

Calling `increment` will increase the value of `counter` by 1 and then return its current value. Because `counter` is a global variable, the change will be persisted to the heap, and subsequent query and update calls will have access to the new `counter` value.

Because the Internet Computer (IC) persists changes with certain fault tolerance guarantees, update calls are executed on many nodes and go through consensus. This leads to latencies of ~2-5 seconds per update call.

Due to the latency and other expenses involved with update methods, it is best to use them only when necessary. Look at the following example:

```
from kybra import query, update, void

message = ""


@query
def get_message() -> str:
    return message


@update
def set_message(new_message: str) -> void:
    global message
    message = new_message
```

You'll notice that we use an update method, `set_message`, only to perform the change to the global `message` variable. We use `get_message`, a query method, to read the message.

Keep in mind that the heap is limited to 4 GiB, and thus there is an upper bound to global variable storage capacity. You can imagine how a simple database like the following would eventually run out of memory with too many entries:

```
from kybra import Opt, query, update, void

db: dict[str, str] = {}


@query
def get(key: str) -> Opt[str]:
    return db.get(key)


@update
def set(key: str, value: str) -> void:
    db[key] = value
```

If you need more than 4 GiB of storage, consider taking advantage of the 48 GiB of stable memory. Stable structures like `StableBTreeMap` give you a nice API for interacting with stable memory. These data structures will be covered in more detail later. Here's a simple example:

```python
from kybra import Opt, query, StableBTreeMap, update, void

db = StableBTreeMap[str, str](memory_id=3, max_key_size=10,
max_value_size=10)


@query
def get(key: str) -> Opt[str]:
    return db.get(key)


@update
def set(key: str, value: str) -> void:
    db.insert(key, value)
```

So far we have only seen how state changes can be persisted. State changes can also be discarded by implicit or explicit traps. A trap is an immediate stop to execution with the ability to provide a message to the execution environment.

Traps can be useful for ensuring that multiple operations are either all completed or all disregarded, or in other words atomic. Keep in mind that these guarantees do not hold once cross-canister calls are introduced, but that's a more advanced topic covered later.

Here's an example of how to trap and ensure atomic changes to your database:

```
from kybra import ic, Opt, query, Record, StableBTreeMap, update, Vec, void


class Entry(Record):
    key: str
    value: str


db = StableBTreeMap[str, str](memory_id=3, max_key_size=10,
max_value_size=10)


@query
def get(key: str) -> Opt[str]:
    return db.get(key)


@update
def set(key: str, value: str) -> void:
    db.insert(key, value)


@update
def set_many(entries: Vec[Entry]) -> void:
    for entry in entries:
        if entry["key"] == "trap":
            ic.trap("explicit trap")

        db.insert(entry["key"], entry["value"])
```

In addition to `ic.trap`, an explicit Python `raise` or any unhandled exception will also trap.

There is a limit to how much computation can be done in a single call to an update method. The current update call limit is 20 billion Wasm instructions. If we modify our database example, we can introduce an update method that runs the risk of reaching the limit:

```
from kybra import nat64, Opt, query, Record, StableBTreeMap, update, void


class Entry(Record):
    key: str
    value: str


db = StableBTreeMap[str, str](memory_id=3, max_key_size=1_000,
max_value_size=1_000)


@query
def get(key: str) -> Opt[str]:
    return db.get(key)


@update
def set(key: str, value: str) -> void:
    db.insert(key, value)


@update
def set_many(num_entries: nat64) -> void:
    for i in range(num_entries):
        db.insert(str(i), str(i))
```

From the `dfx command line` you can call `set_many` like this:

```
dfx canister call my_canister set_many '(100_000)'
```

With an argument of `100_000`, `set_many` will fail with an error `...exceeded the instruction limit for single message execution`.

In terms of update scalability, an individual canister likely has an upper bound of ~900 updates per second.

Candid

Candid is an interface description language created by DFINITY. It can be used to define interfaces between services (canisters), allowing canisters and clients written in various languages to easily interact with each other.

Kybra allows you to express Candid types through a combination of native and Kybra-provided Python types. These types will be necessary in various places as you define your canister. For example, Candid types must be used when defining the parameters and return types of your query and update methods.

It's important to note that the Candid types are represented at runtime using specific Python data structures that may differ in behavior from the description of the actual Candid type. For example, a `float32` Candid type is a Python float, a `nat64` is a Python int, and an `int` is also a Python int.

Keep this in mind as it may result in unexpected behavior. Each Candid type and its equivalent Python runtime value is explained in more detail in this chapter.

A reference of all Candid types available on the Internet Computer (IC) can be found here.

The following is a simple example showing how to import and use most of the Candid types available in Kybra:

```
from kybra import (
    blob,
    float64,
    float32,
    Func,
    int8,
    int16,
    int32,
    int64,
    nat,
    nat8,
    nat16,
    nat32,
    nat64,
    null,
    Opt,
    Principal,
    query,
    Query,
    Record,
    Service,
    service_query,
    service_update,
    Variant,
    Vec,
)


class Candid(Record):
    text: str
    blob: blob
    nat: nat
    nat64: nat64
    nat32: nat32
    nat16: nat16
    nat8: nat8
    int: int
    int64: int64
    int32: int32
    int16: int16
    int8: int8
    float64: float64
    float32: float32
    bool: bool
    null: null
    vec: Vec[str]
    opt: Opt["nat"]
    record: "CandidRecord"
    variant: "CandidVariant"
    func: "CandidFunc"
    service: "MyService"
    principal: Principal


class CandidRecord(Record):
    first_name: str
    last_name: str
    age: nat8


class CandidVariant(Variant, total=False):
    Tag1: null
    Tag2: null
    Tag3: int


class MyService(Service):
    @service_query
    def query1(self) -> bool:
        ...

    @service_update
    def update1(self) -> str:
        ...


CandidFunc = Func(Query[[], Candid])


@query
def candid_types() -> Candid:
    return {
        "text": "text",
        "blob": bytes(),
        "nat": 340_282_366_920_938_463_463_374_607_431_768_211_455,
        "nat64": 18_446_744_073_709_551_615,
        "nat32": 4_294_967_295,
        "nat16": 65_535,
        "nat8": 255,
        "int": 170_141_183_460_469_231_731_687_303_715_884_105_727,
        "int64": 9_223_372_036_854_775_807,
        "int32": 2_147_483_647,
        "int16": 32_767,
        "int8": 127,
        "float64": 0.0,
        "float32": 0.0,
        "bool": True,
        "null": None,
        "vec": ["has one element"],
        "opt": None,
        "record": {"first_name": "John", "last_name": "Doe", "age": 35},
        "variant": {"Tag1": None},
        "func": (Principal.from_str("rrkah-fqaaa-aaaaa-aaaaq-cai"),
"candid_types"),
        "service": MyService(Principal.from_str("aaaaa-aa")),
        "principal": Principal.from_str("ryjl3-tyaaa-aaaaa-aaaba-cai"),
    }
```

Calling `candid_types` with `dfx` will return:

```
(
  record {
    "int" = 170_141_183_460_469_231_731_687_303_715_884_105_727 : int;
    "nat" = 340_282_366_920_938_463_463_374_607_431_768_211_455 : nat;
    "opt" = null;
    "vec" = vec { "has one element" };
    "service" = service "aaaaa-aa";
    "principal" = principal "ryjl3-tyaaa-aaaaa-aaaba-cai";
    "blob" = vec {};
    "bool" = true;
    "func" = func "rrkah-fqaaa-aaaaa-aaaaq-cai".candid_types;
    "int8" = 127 : int8;
    "nat8" = 255 : nat8;
    "null" = null : null;
    "text" = "text";
    "nat16" = 65_535 : nat16;
    "nat32" = 4_294_967_295 : nat32;
    "nat64" = 18_446_744_073_709_551_615 : nat64;
    "int16" = 32_767 : int16;
    "int32" = 2_147_483_647 : int32;
    "int64" = 9_223_372_036_854_775_807 : int64;
    "variant" = variant { Tag1 };
    "float32" = 0 : float32;
    "float64" = 0 : float64;
    "record" = record {
      age = 35 : nat8;
      first_name = "John";
      last_name = "Doe";
    };
  },
)
```

text

The Python type `str` and the Kybra type `text` both correspond to the Candid type text and will become a Python str at runtime.

Python:

```python
from kybra import ic, query


@query
def get_string() -> str:
    return "Hello world!"


@query
def print_string(string: str) -> str:
    ic.print(type(string))
    return string
```

Candid:

```
service: {
    "get_string": () -> (text) query;
    "print_string": (text) -> (text) query;
}
```

blob

The Kybra type `blob` corresponds to the Candid type blob and will become a Python bytes at runtime.

Python:

```
from kybra import blob, ic, query


@query
def get_blob() -> blob:
    return bytes([68, 73, 68, 76, 0, 0])


@query
def print_blob(blob: blob) -> blob:
    ic.print(type(blob))
    return blob
```

Candid:

```
service: {
    "get_blob": () -> (blob) query;
    "print_blob": (blob) -> (blob) query;
}
```

nat

The Kybra type `nat` corresponds to the Candid type nat and will become a Python int at runtime.

Python:

```python
from kybra import ic, nat, query


@query
def get_nat() -> nat:
    return 340_282_366_920_938_463_463_374_607_431_768_211_455


@query
def print_nat(nat: nat) -> nat:
    ic.print(type(nat))
    return nat
```

Candid:

```
service: {
    "get_nat": () -> (nat) query;
    "print_nat": (nat) -> (nat) query;
}
```

nat64

The Kybra type `nat64` corresponds to the Candid type nat64 and will become a Python int at runtime.

Python:

```python
from kybra import ic, nat64, query


@query
def get_nat64() -> nat64:
    return 18_446_744_073_709_551_615


@query
def print_nat64(nat64: nat64) -> nat64:
    ic.print(type(nat64))
    return nat64
```

Candid:

```
service: {
    "get_nat64": () -> (nat64) query;
    "print_nat64": (nat64) -> (nat64) query;
}
```

nat32

The Kybra type `nat32` corresponds to the Candid type nat32 and will become a Python int at runtime.

Python:

```
from kybra import ic, nat32, query


@query
def get_nat32() -> nat32:
    return 4_294_967_295


@query
def print_nat32(nat32: nat32) -> nat32:
    ic.print(type(nat32))
    return nat32
```

Candid:

```
service: {
    "get_nat32": () -> (nat32) query;
    "print_nat32": (nat32) -> (nat32) query;
}
```

nat16

The Kybra type `nat16` corresponds to the Candid type nat16 and will become a Python int at runtime.

Python:

```
from kybra import ic, nat16, query


@query
def get_nat16() -> nat16:
    return 65_535


@query
def print_nat16(nat16: nat16) -> nat16:
    ic.print(type(nat16))
    return nat16
```

Candid:

```
service: {
    "get_nat16": () -> (nat16) query;
    "print_nat16": (nat16) -> (nat16) query;
}
```

nat8

The Kybra type `nat8` corresponds to the Candid type nat8 and will become a Python int at runtime.

Python:

```python
from kybra import ic, nat8, query


@query
def get_nat8() -> nat8:
    return 255


@query
def print_nat8(nat8: nat8) -> nat8:
    ic.print(type(nat8))
    return nat8
```

Candid:

```
service: {
    "get_nat8": () -> (nat8) query;
    "print_nat8": (nat8) -> (nat8) query;
}
```

int

The Kybra type `int` corresponds to the Candid type int and will become a Python int at runtime.

Python:

```
from kybra import ic, query


@query
def get_int() -> int:
    return 170_141_183_460_469_231_731_687_303_715_884_105_727


@query
def print_int(int: int) -> int:
    ic.print(type(int))
    return int
```

Candid:

```
service: {
    "get_int": () -> (int) query;
    "print_int": (int) -> (int) query;
}
```

int64

The Kybra type `int64` corresponds to the Candid type int64 and will become a Python int at runtime.

Python:

```
from kybra import ic, int64, query


@query
def get_int64() -> int64:
    return 9_223_372_036_854_775_807


@query
def print_int64(int64: int64) -> int64:
    ic.print(type(int64))
    return int64
```

Candid:

```
service: {
    "get_int64": () -> (int64) query;
    "print_int64": (int64) -> (int64) query;
}
```

int32

The Kybra type `int32` corresponds to the Candid type int32 and will become a Python int at runtime.

Python:

```
from kybra import ic, int32, query


@query
def get_int32() -> int32:
    return 2_147_483_647


@query
def print_int32(int32: int32) -> int32:
    ic.print(type(int32))
    return int32
```

Candid:

```
service: {
    "get_int32": () -> (int32) query;
    "print_int32": (int32) -> (int32) query;
}
```

int16

The Kybra type `int16` corresponds to the Candid type int16 and will become a Python int at runtime.

Python:

```
from kybra import ic, int16, query


@query
def get_int16() -> int16:
    return 32_767


@query
def print_int16(int16: int16) -> int16:
    ic.print(type(int16))
    return int16
```

Candid:

```
service: {
    "get_int16": () -> (int16) query;
    "print_int16": (int16) -> (int16) query;
}
```

int8

The Kybra type `int8` corresponds to the Candid type int8 and will become a Python int at runtime.

Python:

```
from kybra import ic, int8, query


@query
def get_int8() -> int8:
    return 127


@query
def print_int8(int8: int8) -> int8:
    ic.print(type(int8))
    return int8
```

Candid:

```
service: {
    "get_int8": () -> (int8) query;
    "print_int8": (int8) -> (int8) query;
}
```

float64

The Kybra type `float64` corresponds to the Candid type float64 and will become a Python float at runtime.

Python:

```python
import math

from kybra import float64, ic, query


@query
def get_float64() -> float64:
    return math.e


@query
def print_float64(float64: float64) -> float64:
    ic.print(type(float64))
    return float64
```

Candid:

```
service: {
    "get_float64": () -> (float64) query;
    "print_float64": (float64) -> (float64) query;
}
```

float32

The Kybra type `float32` corresponds to the Candid type float32 and will become a Python float at runtime.

Python:

```python
import math

from kybra import float32, ic, query


@query
def get_float32() -> float32:
    return math.pi


@query
def print_float32(float32: float32) -> float32:
    ic.print(type(float32))
    return float32
```

Candid:

```
service: {
    "get_float32": () -> (float32) query;
    "print_float32": (float32) -> (float32) query;
}
```

bool

The Python type `bool` corresponds to the Candid type bool and will become a Python Boolean Value at runtime.

Python:

```
from kybra import ic, query


@query
def get_bool() -> bool:
    return True


@query
def print_bool(bool: bool) -> bool:
    ic.print(type(bool))
    return bool
```

Candid:

```
service: {
    "get_bool": () -> (bool) query;
    "print_bool": (bool) -> (bool) query;
}
```

null

The Kybra type `null` corresponds to the Candid type null and will become the Python Null Object at runtime.

Python:

```
from kybra import ic, null, query


@query
def get_null() -> null:
    return None


@query
def print_null(none: null) -> null:
    ic.print(type(none))
    return none
```

Candid:

```
service: {
    "get_null": () -> (null) query;
    "print_null": (null) -> (null) query;
}
```

vec

The Kybra type `Vec` corresponds to the Candid type vec and will become an array of the specified type at runtime.

Python:

```python
from kybra import int32, query, Vec


@query
def get_numbers() -> Vec[int32]:
    return [0, 1, 2, 3]
```

Candid:

```
service: {
    "get_numbers": () -> (vec int32) query;
}
```

Opt

The Kybra type `Opt` corresponds to the Candid type opt and will become the enclosed Python type or None at runtime.

Python:

```python
from kybra import Opt, query


@query
def get_opt_some() -> Opt[bool]:
    return True


@query
def get_opt_none() -> Opt[bool]:
    return None
```

Candid:

```
service: {
    "get_opt_some": () -> (opt bool) query;
    "get_opt_none": () -> (opt bool) query;
}
```

record

Python classes that inherit from the Kybra type `Record` correspond to the Candid record type and will become Python TypedDicts at runtime.

Python:

```python
from kybra import Record, Vec


class Post(Record):
    id: str
    author: "User"
    text: str
    thread: "Thread"


class Thread(Record):
    id: str
    author: "User"
    posts: Vec[Post]
    title: str


class User(Record):
    id: str
    posts: Vec[Post]
    thread: Vec[Thread]
    username: str
```

Candid:

```
type Post = record {
    "id": text;
    "author": User;
    "text": text;
    "thread": Thread;
};

type Thread = record {
    "id": text;
    "author": User;
    "posts": vec Post;
    "title": text;
};

type User = record {
    "id": text;
    "posts": vec Post;
    "threads": vec Thread;
    "username": text;
};
```

variant

Python classes that inherit from the Kybra type `Variant` correspond to the Candid variant type and will become Python TypedDicts at runtime.

Python:

```python
from kybra import nat32, null, Variant


class ReactionType(Variant, total=False):
    Fire: null
    ThumbsUp: null
    ThumbsDown: null
    Emotion: "Emotion"
    Firework: "Firework"


class Emotion(Variant, total=False):
    Happy: null
    Sad: null


class Firework(Variant, total=False):
    Color: str
    NumStreaks: nat32
```

Candid:

```
type ReactionType = variant {
    "Fire": null;
    "ThumbsUp": null;
    "ThumbsDown": null;
    "Emotion": Emotion;
    "Firework": Firework
};

type Emotion = variant {
    "Happy": null;
    "Sad": null
};

type Firework = record {
    "Color": text;
    "NumStreaks": nat32;
};
```

func

The Kybra type `Func` corresponds to the Candid type func and at runtime will become a Python tuple with two elements, the first being an ic-py Principal and the second being a Python str. The `ic-py Principal` represents the `principal` of the canister/service where the function exists, and the `str` represents the function's name.

Python:

```
from kybra import Func, nat64, null, Principal, query, Query, Record, Update,
Variant


class User(Record):
    id: str
    basic_func: "BasicFunc"
    complex_func: "ComplexFunc"


class Reaction(Variant, total=False):
    Good: null
    Bad: null
    BasicFunc: "BasicFunc"
    ComplexFunc: "ComplexFunc"


BasicFunc = Func(Query[[str], str])
ComplexFunc = Func(Update[[User, Reaction], nat64])


@query
def get_basic_func() -> BasicFunc:
    return (Principal.from_str("rrkah-fqaaa-aaaaa-aaaaq-cai"),
"simple_function_name")


@query
def get_complex_func() -> ComplexFunc:
    return (Principal.from_str("ryjl3-tyaaa-aaaaa-aaaba-cai"),
"complex_function_name")
```

Candid:

```
type User = record {
    "id": text;
    "basic_func": BasicFunc;
    "complex_func": ComplexFunc;
};
type Reaction = variant { "Good": null; "Bad": null; "BasicFunc": BasicFunc;
"ComplexFunc": ComplexFunc };

type BasicFunc = func (text) -> (text) query;
type ComplexFunc = func (User, Reaction) -> (nat64);

service: () -> {
    "get_basic_func": () -> (BasicFunc) query;
    "get_complex_func": () -> (ComplexFunc) query;
}
```

service

Python classes that inherit from the Kybra type `Service` correspond to the Candid service type and will become child classes capable of creating instances that can perform cross-canister calls at runtime.

Python:

```
from kybra import (
    Async,
    CallResult,
    Principal,
    query,
    Service,
    service_query,
    service_update,
    update,
)


class SomeService(Service):
    @service_query
    def query1(self) -> bool:
        ...

    @service_update
    def update1(self) -> str:
        ...


@query
def get_service() -> SomeService:
    return SomeService(Principal.from_str("aaaaa-aa"))


@update
def call_service(service: SomeService) -> Async[str]:
    result: CallResult[str] = yield service.update1()

    if result.Err is not None:
        raise Exception(f"call to service.update1 failed with: {result.Err}")

    return result.Ok
```

Candid:

```
service { query1 : () -> (bool) query; update1 : () -> (text) }
```

principal

The Kybra type `Principal` corresponds to the Candid type principal and will become an ic-py Principal at runtime.

Python:

```
from kybra import ic, Principal, query


@query
def get_principal() -> Principal:
    return Principal.from_str("rrkah-fqaaa-aaaaa-aaaaq-cai")


@query
def print_principal(principal: Principal) -> Principal:
    ic.print(type(principal))
    return principal
```

Candid:

```
service: {
    "get_principal": () -> (principal) query;
    "print_principal": (principal) -> (principal) query;
}
```

reserved

The Kybra type `reserved` corresponds to the Candid type reserved and will become the Python Null Object at runtime.

Python:

```
from kybra import ic, query, reserved


@query
def get_reserved() -> reserved:
    return "anything"


@query
def print_reserved(reserved: reserved) -> reserved:
    ic.print(type(reserved))
    return reserved
```

Candid:

```
service: {
    "get_reserved": () -> (reserved) query;
    "print_reserved": (reserved) -> (reserved) query;
}
```

empty

The Kybra type `empty` corresponds to the Candid type empty and has no Python value at runtime.

Python:

```
from kybra import empty, ic, query


@query
def get_empty() -> empty:
    raise Exception("Anything you want")


# Note: It is impossible to call this function because it requires an
argument
# but there is no way to pass an "empty" value as an argument.
@query
def print_empty(empty: empty) -> empty:
    ic.print(type(empty))
    raise Exception("Anything you want")
```

Candid:

```
service: {
    "get_empty": () -> (empty) query;
    "print_empty": (empty) -> (empty) query;
}
```

Stable Structures

TLDR

- 48 GiB of stable memory
- Persistent across upgrades
- Familiar API
- Must specify memory id
- Must specify maximum key size
- Must specify maximum value size
- No migrations per memory id

Stable structures are data structures with familiar APIs that allow access to stable memory. Stable memory is a separate memory location from the heap that currently allows up to 48 GiB of storage. Stable memory persists automatically across upgrades.

Persistence on the Internet Computer (IC) is very important to understand. When a canister is upgraded (its code is changed after being initially deployed) its heap is wiped. This includes all global variables.

On the other hand, anything stored in stable memory will be preserved. Writing and reading to and from stable memory can be done with a low-level API, but it is generally easier and preferable to use stable structures.

Kybra currently provides one stable structure called `StableBTreeMap`. It's similar to a Python dictionary or a map that you'd find in most languages, and has most of the common operations you'd expect such as reading, inserting, and removing values.

Here's how to define a simple `StableBTreeMap`. Each `StableBTreeMap` must be defined in the global scope (not within any functions or objects etc):

```
from kybra import nat8, StableBTreeMap

map = StableBTreeMap[nat8, str](memory_id=3, max_key_size=100,
max_value_size=1_000)
```

This is a `StableBTreeMap` with a key of type `nat8` and a value of type `str`. Key and value types can be any Candid type.

This `StableBTreeMap` also has a `memory id` of `3`, a maximum key size of `100` bytes and a maximum value size of `1_000` bytes. You must statically specify the `memory id`, maximum key size, and maximum value sizes (they cannot be variables).

The `memory id` can be a number between `0` and `254`, but `memory ids` `0`, `1`, `2`, `252`, `253`, and `254` are currently reserved. We hope to reduce the complexity around `memory id` in the future, and perhaps remove the need to specify it entirely.

Each `StableBTreeMap` instance must have a unique `memory id`. Once a `memory id` is allocated, it cannot be used with a different `StableBTreeMap`. This means you can't create another `StableBTreeMap` using the same `memory id`, and you can't change the key or value types of an existing `StableBTreeMap`. This problem will be addressed.

Here's an example showing all of the basic `StableBTreeMap` operations:

```
from kybra import (
    Alias,
    nat64,
    nat8,
    Opt,
    query,
    StableBTreeMap,
    Tuple,
    update,
    Vec,
)

Key = Alias[nat8]
Value = Alias[str]


map = StableBTreeMap[Key, Value](memory_id=3, max_key_size=100,
max_value_size=1_000)


@query
def contains_key(key: Key) -> bool:
    return map.contains_key(key)


@query
def get(key: Key) -> Opt[Value]:
    return map.get(key)


@update
def insert(key: Key, value: Value) -> Opt[Value]:
    return map.insert(key, value)


@query
def is_empty() -> bool:
    return map.is_empty()


@query
def items() -> Vec[Tuple[Key, Value]]:
    return map.items()


@query
def keys() -> Vec[Key]:
    return map.keys()


@query
def len() -> nat64:
    return map.len()


@update
```

```
def remove(key: Key) -> Opt[Value]:
    return map.remove(key)


@query
def values() -> Vec[Value]:
    return map.values()
```

With these basic operations you can build more complex CRUD database applications:

```
import secrets

from kybra import (
    blob,
    ic,
    nat64,
    Opt,
    Principal,
    query,
    Record,
    StableBTreeMap,
    update,
    Variant,
    Vec,
)


class User(Record):
    id: Principal
    created_at: nat64
    recording_ids: Vec[Principal]
    username: str


class Recording(Record):
    id: Principal
    audio: blob
    created_at: nat64
    name: str
    user_id: Principal


users = StableBTreeMap[Principal, User](
    memory_id=3, max_key_size=38, max_value_size=100_000
)
recordings = StableBTreeMap[Principal, Recording](
    memory_id=4, max_key_size=38, max_value_size=5_000_000
)


@update
def create_user(username: str) -> User:
    id = generate_id()
    user: User = {
        "id": id,
        "created_at": ic.time(),
        "recording_ids": [],
        "username": username,
    }

    users.insert(user["id"], user)

    return user


@query
```

```
def read_users() -> Vec[User]:
    return users.values()


@query
def read_user_by_id(id: Principal) -> Opt[User]:
    return users.get(id)


class DeleteUserResult(Variant, total=False):
    Ok: User
    Err: "DeleteUserErr"


class DeleteUserErr(Variant, total=False):
    UserDoesNotExist: Principal


@update
def delete_user(id: Principal) -> DeleteUserResult:
    user = users.get(id)

    if user is None:
        return {"Err": {"UserDoesNotExist": id}}

    for recording_id in user["recording_ids"]:
        recordings.remove(recording_id)

    users.remove(user["id"])

    return {"Ok": user}


class CreateRecordingResult(Variant, total=False):
    Ok: Recording
    Err: "CreateRecordingErr"


class CreateRecordingErr(Variant, total=False):
    UserDoesNotExist: Principal


@update
def create_recording(
    audio: blob, name: str, user_id: Principal
) -> CreateRecordingResult:
    user = users.get(user_id)

    if user is None:
        return {"Err": {"UserDoesNotExist": user_id}}

    id = generate_id()
    recording: Recording = {
        "id": id,
        "audio": audio,
        "created_at": ic.time(),
        "name": name,
        "user_id": user_id,
    }

    recordings.insert(recording["id"], recording)

    updated_user: User = {
        "id": user["id"],
        "created_at": user["created_at"],
        "username": user["username"],
        "recording_ids": [*user["recording_ids"], recording["id"]],
    }

    users.insert(updated_user["id"], updated_user)

    return {"Ok": recording}


@query
def read_recordings() -> Vec[Recording]:
    return recordings.values()


@query
def read_recording_by_id(id: Principal) -> Opt[Recording]:
    return recordings.get(id)


class DeleteRecordingResult(Variant, total=False):
    Ok: Recording
    Err: "DeleteRecordingError"


class DeleteRecordingError(Variant, total=False):
    RecordingDoesNotExist: Principal
    UserDoesNotExist: Principal


@update
def delete_recording(id: Principal) -> DeleteRecordingResult:
    recording = recordings.get(id)

    if recording is None:
        return {"Err": {"RecordingDoesNotExist": id}}

    user = users.get(recording["user_id"])

    if user is None:
        return {"Err": {"UserDoesNotExist": recording["user_id"]}}

    updated_user: User = {
        "id": user["id"],
        "created_at": user["created_at"],
        "username": user["username"],
        "recording_ids": list(
            filter(
                lambda recording_id: recording_id.to_str() !=
recording["id"].to_str(),
                user["recording_ids"],
            )
        ),
    }

    users.insert(updated_user["id"], updated_user)

    recordings.remove(id)

    return {"Ok": recording}


def generate_id() -> Principal:
    random_bytes = secrets.token_bytes(29)

    return Principal.from_hex(random_bytes.hex())
```

The example above shows a very basic audio recording backend application. There are two types of entities that need to be stored, `User` and `Recording`. These are represented as `Candid` records.

Each entity gets its own `StableBTreeMap`:

```
from kybra import blob, nat64, Principal, Record, StableBTreeMap, Vec


class User(Record):
    id: Principal
    created_at: nat64
    recording_ids: Vec[Principal]
    username: str


class Recording(Record):
    id: Principal
    audio: blob
    created_at: nat64
    name: str
    user_id: Principal


users = StableBTreeMap[Principal, User](
    memory_id=3, max_key_size=38, max_value_size=100_000
)
recordings = StableBTreeMap[Principal, Recording](
    memory_id=4, max_key_size=38, max_value_size=5_000_000
)
```

Notice that each `StableBTreeMap` has a unique `memory id`. The maximum key and value sizes are also set according to the expected application usage.

You can figure out the appropriate maximum key and value sizes by reasoning about your application and engaging in some trial and error using the `insert` method. Calling `insert` on a `StableBTreeMap` will throw an error which in some cases will have the information that you need to determine the maximum key or value size.

If you attempt to insert a key or value that is too large, the `KeyTooLarge` and `ValueTooLarge` errors will show you the size of the value that you attempted to insert. You can increase the maximum key or value size based on the information you receive from the `KeyTooLarge` and `ValueTooLarge` errors and try inserting again.

Thus through some trial and error you can whittle your way to a correct solution. In some cases all of your values will have an obvious static maximum size. In the audio recording example, trial and error revealed that `Principal` is most likely always 38 bytes, thus the maximum key size is set to 38.

Maximum value sizes can be more tricky to figure out, especially if the values are records or variants with dynamic fields such as arrays. `User` has one such dynamic field, `recording_ids`. Since each recording id is a `Principal`, we know that each will take up 38 bytes. The other fields on `User` shouldn't take up too many bytes so we'll ignore them for our analysis.

We've set the maximum value size of `User` to be 100_000 bytes. If we divide 100_00 by 38, we get ~2_631. This will result in each user being able to store around that many recordings. That's acceptable for our example, and so we'll go with it.

As for `Recording`, the largest dynamic field is `audio`, which will be the actual bytes of the audio recording. We've set the maximum value size here to 5_000_000, which should allow for recordings of ~5 MB in size. That seems reasonable for our example, and so we'll go with it.

As you can see, finding the correct maximum key and value sizes is a bit of an art right now. Combining some trial and error with reasoning about your specific application should get you a working solution in most cases. It's our hope that the need to specify maximum key and value sizes will be removed in the future.

Caveats

memory ids

The `memory id` can be a number between `0` and `254`, but `memory ids` `0`, `1`, `2`, `252`, `253`, and `254` are currently reserved. We hope to reduce the complexity around `memory id` in the future, and perhaps remove the need for it entirely.

Keys

You should be wary when using `float64`, `float32`, `service`, or `func` in any type that is a key for a stable structure. These types do not have the ability to be strictly ordered in all cases. `service` and `func` will have no order. `float64` and `float32` will treat `NaN` as less than any other type. These caveats may impact key performance.

Cross-canister

Examples:

- bitcoin
- composite_queries
- cross_canister_calls
- cycles
- ethereum_json_rpc
- func_types
- heartbeat
- generators
- ledger_canister
- management_canister
- outgoing_http_requests
- threshold_ecdsa
- rejections
- timers
- whoami

Canisters are generally able to call the query or update methods of other canisters in any subnet. We refer to these types of calls as cross-canister calls.

A cross-canister call begins with a definition of the canister to be called, referred to as a service.

Imagine a simple service called `token_canister`:

```python
from kybra import ic, nat64, Principal, StableBTreeMap, update

accounts = StableBTreeMap[Principal, nat64](
    memory_id=3, max_key_size=38, max_value_size=15
)


@update
def transfer(to: Principal, amount: nat64) -> nat64:
    from_ = ic.caller()

    from_balance = accounts.get(from_) or 0
    to_balance = accounts.get(to) or 0

    accounts.insert(from_, from_balance - amount)
    accounts.insert(to, to_balance + amount)

    return amount
```

Here's how you would create its service definition:

```
from kybra import nat64, Principal, Service, service_update


class TokenCanister(Service):
    @service_update
    def transfer(self, to: Principal, amount: nat64) -> nat64:
        ...
```

Once you have a service definition you can instantiate it with the service's `Principal` and then invoke its methods.

Here's how to instantiate `TokenCanister` :

```
token_canister = TokenCanister(
    Principal.from_str('r7inp-6aaaa-aaaaa-aaabq-cai')
)
```

And here's a more complete example of a service called `payout_canister` that performs a cross-canister call to `token_canister` :

```python
from kybra import (
    Async,
    CallResult,
    match,
    nat64,
    Principal,
    Service,
    service_update,
    update,
    Variant,
)


class TokenCanister(Service):
    @service_update
    def transfer(self, to: Principal, amount: nat64) -> nat64:
        ...


token_canister = TokenCanister(Principal.from_str("r7inp-6aaaa-aaaaa-aaabq-
cai"))


class PayoutResult(Variant, total=False):
    Ok: nat64
    Err: str


@update
def payout(to: Principal, amount: nat64) -> Async[PayoutResult]:
    result: CallResult[nat64] = yield token_canister.transfer(to, amount)

    return match(result, {"Ok": lambda ok: {"Ok": ok}, "Err": lambda err:
{"Err": err}})
```

Notice that the `token_canister.transfer` method, because it is a cross-canister method, returns a `CallResult`. All cross-canister calls return `CallResult`, which has an `Ok` or `Err` property depending on if the cross-canister call was successful or not.

The IC guarantees that cross-canister calls will return. This means that, generally speaking, you will always receive a `CallResult`. Kybra does not raise on cross-canister calls. Wrapping your cross-canister call in a `try...except` most likely won't do anything useful.

Let's add to our example code and explore adding some practical result-based error-handling to stop people from stealing tokens.

`token_canister`:

```python
from kybra import ic, nat64, Principal, StableBTreeMap, update, Variant

accounts = StableBTreeMap[Principal, nat64](
    memory_id=3, max_key_size=38, max_value_size=15
)


class TransferResult(Variant, total=False):
    Ok: nat64
    Err: "TransferError"


class TransferError(Variant, total=False):
    InsufficientBalance: nat64


@update
def transfer(to: Principal, amount: nat64) -> TransferResult:
    from_ = ic.caller()

    from_balance = accounts.get(from_) or 0

    if from_balance < amount:
        return {"Err": {"InsufficientBalance": from_balance}}

    to_balance = accounts.get(to) or 0

    accounts.insert(from_, from_balance - amount)
    accounts.insert(to, to_balance + amount)

    return {"Ok": amount}
```

payout_canister:

```
from kybra import (
    Async,
    CallResult,
    match,
    nat64,
    Principal,
    Service,
    service_update,
    update,
    Variant,
)


class TokenCanister(Service):
    @service_update
    def transfer(self, to: Principal, amount: nat64) -> "TransferResult":
        ...


class TransferResult(Variant, total=False):
    Ok: nat64
    Err: "TransferError"


class TransferError(Variant, total=False):
    InsufficientBalance: nat64


token_canister = TokenCanister(Principal.from_str("r7inp-6aaaa-aaaaa-aaabq-
cai"))


class PayoutResult(Variant, total=False):
    Ok: nat64
    Err: str


@update
def payout(to: Principal, amount: nat64) -> Async[PayoutResult]:
    call_result: CallResult[TransferResult] = yield
token_canister.transfer(to, amount)

    def handle_transfer_result_ok(transfer_result: TransferResult) ->
PayoutResult:
        return match(
            transfer_result,
            {
                "Ok": lambda ok: {"Ok": ok},
                "Err": lambda err: {"Err": str(err)},
            },
        )

    return match(
        call_result,
        {
            "Ok": handle_transfer_result_ok,
            "Err": lambda err: {"Err": err},
        },
    )
```

So far we have only shown a cross-canister call from an update method. Update methods can call other update methods or query methods (but not composite query methods as discussed below). If an update method calls a query method, that query method will be called in replicated mode. Replicated mode engages the consensus process, but for queries the state will still be discarded.

Cross-canister calls can also be initiated from query methods. These are known as composite queries, and in Kybra they are simply query methods that return a generator using the `Async` type. Composite queries can call other composite query methods and regular query methods. Composite queries cannot call update methods.

Here's an example of a composite query method:

```
from kybra import (
    Async,
    CallResult,
    match,
    Principal,
    query,
    Service,
    service_query,
    Variant,
)


class SomeCanister(Service):
    @service_query
    def query_for_boolean(self) -> bool:
        ...


some_canister = SomeCanister(Principal.from_str("ryjl3-tyaaa-aaaaa-aaaba-
cai"))


class QuerySomeCanisterResult(Variant, total=False):
    Ok: bool
    Err: str


@query
def query_some_canister() -> Async[QuerySomeCanisterResult]:
    call_result: CallResult[bool] = yield some_canister.query_for_boolean()

    return match(
        call_result, {"Ok": lambda ok: {"Ok": ok}, "Err": lambda err: {"Err":
err}}
    )
```

You can expect cross-canister calls within the same subnet to take up to a few seconds to complete, and cross-canister calls across subnets take about double that time.

If you don't need to wait for your cross-canister call to return, you can use `notify`:

```
from kybra import (
    null,
    Principal,
    query,
    RejectionCode,
    Service,
    service_update,
    Variant,
    void,
)


class SomeCanister(Service):
    @service_update
    def receive_notification(self) -> void:
        ...


some_canister = SomeCanister(Principal.from_str("ryjl3-tyaaa-aaaaa-aaaba-
cai"))


class ReceiveNotificationResult(Variant, total=False):
    Ok: null
    Err: RejectionCode


@query
def send_notification() -> ReceiveNotificationResult:
    return some_canister.receive_notification().notify()
```

If you need to send cycles with your cross-canister call, you can call `with_cycles` before calling `call` or `notify`:

```
from kybra import (
    null,
    Principal,
    query,
    RejectionCode,
    Service,
    service_update,
    Variant,
    void,
)


class SomeCanister(Service):
    @service_update
    def receive_notification(self) -> void:
        ...


some_canister = SomeCanister(Principal.from_str("ryjl3-tyaaa-aaaaa-aaaba-
cai"))


class ReceiveNotificationResult(Variant, total=False):
    Ok: null
    Err: RejectionCode


@query
def send_notification() -> ReceiveNotificationResult:
    return
some_canister.receive_notification().with_cycles(1_000_000).notify()
```

HTTP

This chapter is a work in progress.

Incoming HTTP requests

Examples:

- http_counter

```python
from kybra import blob, Func, nat16, Opt, query, Query, Record, Tuple,
Variant, Vec


class HttpRequest(Record):
    method: str
    url: str
    headers: Vec["Header"]
    body: blob


class HttpResponse(Record):
    status_code: nat16
    headers: Vec["Header"]
    body: blob
    streaming_strategy: Opt["StreamingStrategy"]
    upgrade: Opt[bool]


Header = Tuple[str, str]


class StreamingStrategy(Variant):
    Callback: "CallbackStrategy"


class CallbackStrategy(Record):
    callback: "Callback"
    token: "Token"


Callback = Func(Query[["Token"], "StreamingCallbackHttpResponse"])


class StreamingCallbackHttpResponse(Record):
    body: blob
    token: Opt["Token"]


class Token(Record):
    arbitrary_data: str


@query
def http_request(req: HttpRequest) -> HttpResponse:
    return {
        "status_code": 200,
        "headers": [],
        "body": bytes(),
        "streaming_strategy": None,
        "upgrade": False,
    }
```

Outgoing HTTP requests

Examples:

- ethereum_json_rpc
- outgoing_http_requests

```
from kybra import (
    Alias,
    Async,
    CallResult,
    ic,
    match,
    init,
    nat32,
    query,
    StableBTreeMap,
    update,
    void,
)
from kybra.canisters.management import (
    HttpResponse,
    HttpTransformArgs,
    management_canister,
)

JSON = Alias[str]


stable_storage = StableBTreeMap[str, str](
    memory_id=3, max_key_size=20, max_value_size=1_000
)


@init
def init_(ethereum_url: str) -> void:
    stable_storage.insert("ethereum_url", ethereum_url)


@update
def eth_get_balance(ethereum_address: str) -> Async[JSON]:
    http_result: CallResult[HttpResponse] = yield
management_canister.http_request(
        {
            "url": stable_storage.get("ethereum_url") or "",
            "max_response_bytes": 2_000,
            "method": {"post": None},
            "headers": [],
            "body": f'{{"jsonrpc":"2.0","method":"eth_getBalance","params":
["{ethereum_address}","earliest"],"id":1}}'.encode(
                "utf-8"
            ),
            "transform": {"function": (ic.id(), "eth_transform"), "context":
bytes()},
        }
    ).with_cycles(50_000_000)

    return match(
        http_result,
        {"Ok": lambda ok: ok["body"].decode("utf-8"), "Err": lambda err:
ic.trap(err)},
    )
```

```
@update
def eth_get_block_by_number(number: nat32) -> Async[JSON]:
    http_result: CallResult[HttpResponse] = yield
management_canister.http_request(
        {
            "url": stable_storage.get("ethereum_url") or "",
            "max_response_bytes": 2_000,
            "method": {"post": None},
            "headers": [],
            "body":
f'{{"jsonrpc":"2.0","method":"eth_getBlockByNumber","params":
["{hex(number)}", false],"id":1}}'.encode(
                "utf-8"
            ),
            "transform": {"function": (ic.id(), "eth_transform"), "context":
bytes()},
        }
    ).with_cycles(50_000_000)

    return match(
        http_result,
        {"Ok": lambda ok: ok["body"].decode("utf-8"), "Err": lambda err:
ic.trap(err)},
    )


@query
def eth_transform(args: HttpTransformArgs) -> HttpResponse:
    http_response = args["response"]

    http_response["headers"] = []

    return http_response
```

Management Canister

This chapter is a work in progress.

You can access the management canister like this:

```
from kybra import Async, blob, CallResult, match, update, Variant
from kybra.canisters.management import management_canister


class RandomBytesResult(Variant, total=False):
    Ok: blob
    Err: str


@update
def random_bytes() -> Async[RandomBytesResult]:
    call_result: CallResult[blob] = yield management_canister.raw_rand()

    return match(
        call_result, {"Ok": lambda ok: {"Ok": ok}, "Err": lambda err: {"Err":
err}}
    )
```

See the management canister types for all methods and their parameter and return types.

Canister Lifecycle

This chapter is a work in progress.

```
from kybra import ic, init, post_upgrade, pre_upgrade, void


@init
def init_() -> void:
    ic.print("runs on first canister install")


@pre_upgrade
def pre_upgrade_() -> void:
    ic.print("runs before canister upgrade")


@post_upgrade
def post_upgrade_() -> void:
    ic.print("runs after canister upgrade")
```

Caveats

params

Keep in mind this caveat when working with `init` or `post_upgrade` params.

ic.caller()

Usually `ic.caller()` if called from `init` or `post_upgrade` is the principal of your local dfx identity. In Kybra canisters `ic.caller()` if called from `init` or `post_upgrade` is the canister's own principal.

guard functions

`init` and `post_upgrade` cannot have guard functions applied to them.

Timers

This chapter is a work in progress.

```
from kybra import (
    Async,
    blob,
    CallResult,
    Duration,
    ic,
    match,
    nat8,
    query,
    Record,
    TimerId,
    update,
    void,
)
from kybra.canisters.management import management_canister


class StatusReport(Record):
    single: bool
    inline: nat8
    capture: str
    repeat: nat8
    single_cross_canister: blob
    repeat_cross_canister: blob


class TimerIds(Record):
    single: TimerId
    inline: TimerId
    capture: TimerId
    repeat: TimerId
    single_cross_canister: TimerId
    repeat_cross_canister: TimerId


status: StatusReport = {
    "single": False,
    "inline": 0,
    "capture": "",
    "repeat": 0,
    "single_cross_canister": bytes(),
    "repeat_cross_canister": bytes(),
}


@update
def clear_timer(timer_id: TimerId) -> void:
    ic.clear_timer(timer_id)
    ic.print(f"timer {timer_id} cancelled")


@update
def set_timers(delay: Duration, interval: Duration) -> TimerIds:
    captured_value = "🚩"

    single_id = ic.set_timer(delay, one_time_timer_callback)
    # Note: You cannot set global variables from within a lambda but you can
    # call a function that sets a global variable. So we've moved the
"setting"
    # functionality out into helper functions while the printing is kept here
in
    # the lambda.

    inline_id = ic.set_timer(
        delay,
        lambda: update_inline_status() or ic.print("Inline timer called"),
    )

    capture_id = ic.set_timer(
        delay,
        lambda: update_capture_status(captured_value)
        or ic.print(f"Timer captured value: {captured_value}"),
    )

    repeat_id = ic.set_timer_interval(interval, repeat_timer_callback)

    single_cross_canister_id = ic.set_timer(delay,
single_cross_canister_timer_callback)

    repeat_cross_canister_id = ic.set_timer_interval(
        interval, repeat_cross_canister_timer_callback
    )

    return {
        "single": single_id,
        "inline": inline_id,
        "capture": capture_id,
        "repeat": repeat_id,
        "single_cross_canister": single_cross_canister_id,
        "repeat_cross_canister": repeat_cross_canister_id,
    }


@query
def status_report() -> StatusReport:
    return status


def one_time_timer_callback():
    status["single"] = True
    ic.print("one_time_timer_callback called")


def repeat_timer_callback():
    status["repeat"] += 1
    ic.print(f"Repeating timer. Call {status['repeat']}")


def update_inline_status():
    status["inline"] = 1
```

```
def update_capture_status(value: str):
    status["capture"] = value


def single_cross_canister_timer_callback() -> Async[void]:
    ic.print("single_cross_canister_timer_callback")

    result: CallResult[blob] = yield management_canister.raw_rand()

    def handle_ok(ok: blob):
        status["single_cross_canister"] = ok

    match(result, {"Ok": handle_ok, "Err": lambda err: ic.print(err)})


def repeat_cross_canister_timer_callback() -> Async[void]:
    ic.print("repeat_cross_canister_timer_callback")

    result: CallResult[blob] = yield management_canister.raw_rand()

    def handle_ok(ok: blob):
        status["repeat_cross_canister"] += ok

    match(result, {"Ok": handle_ok, "Err": lambda err: ic.print(err)})
```

Cycles

This chapter is a work in progress.

Cycles are essentially units of computational resources such as bandwidth, memory, and CPU instructions. Costs are generally metered on the Internet Computer (IC) by cycles. You can see a breakdown of all cycle costs here: https://internetcomputer.org /docs/current/developer-docs/production/computation-and-storage-costs

Currently queries do not have any cycle costs.

Most important to you will probably be update costs.

TODO break down some cycle scenarios maybe? Perhaps we should show some of our analyses for different types of applications. Maybe show how to send and receive cycles, exactly how to do it.

Show all of the APIs for sending or receiving cycles?

Perhaps we don't need to do that here, since each API will show this information.

Maybe here we just show the basic concept of cycles, link to the main cycles cost page, and show a few examples of how to break down these costs or estimate these costs.

Caveats

Unknown security vulnerabilities

Kybra is a beta project using a new Python interpreter. See the disclaimer for more information.

cdk metadata

- All Kybra canisters have a public metadata key-value pair named `cdk` which contains the version of Kybra the canister is using
    - Anyone can query this metadata which may be a security risk

No C extensions

Any PyPI packages or other Python code that relies on C extensions will not currently work. It is a major goal for us to support C extensions in the future.

Performance

Kybra is probably ~7-20x less performant than what you would expect from CPython. We hope to eventually use `CPython` as Kybra's underlying Python interpreter.

init and post_upgrade

under-the-hood nuances

In order to allow for larger Kybra Wasm modules and to enable the Python stdlib, we had to introduce some trade-offs into the `init` and `post_upgrade` processes:

1. The under-the-hood Rust `init` function of your application canister is never called directly, only the `post_upgrade` function. This function will call your Python `init` or

`post_upgrade` function in a timer callback

2. When deploying to a local replica, errors from your `init` or `post_upgrade` will be logged from your replica, not from the `dfx deploy` process
3. When deploying to mainnet, errors from your `init` or `post_upgrade` will not be logged
4. Your canister will go through the following main steps when deploying: first a deployer canister Wasm binary will be deployed to the canister, then your application Wasm binary will be chunk uploaded to the deployer canister along with the Python stdlib, and finally the deployer canister will call `install_code` on itself with the full application Wasm binary

params

If you add parameters to your `init` or `post_upgrade` methods, you will need to manually add these parameters to the service in your Candid file for the first deploy to succeed. Any time you change these parameters, you will need to manually add these changes into your Candid file.

It is also recommended to start clean (`dfx start --clean`) when first deploying after changing these parameters.

Here's an example of a basic Candid file with parameters `text` , `bool` , and `int32` :

```
service : (text, bool, int32) -> {}
```

guard functions

`init` and `post_upgrade` cannot have guard functions applied to them.

No encrypted dfx identities

Kybra currently does not support encrypted identities. If you are asked for a password during `dfx deploy` , you'll need to create a new unencrypted dfx identity:

```
dfx identity new test_unencrypted --storage-mode plaintext
dfx identity use test_unencrypted

dfx deploy
```

No candid:service metadata

Kybra does not put the Candid metadata into `candid:service` automatically. Currently if you want to retrieve the Candid from the canister you can call the `__get_candid_interface_tmp_hack` method.

Wasm module hash now less useful

Due to protocol and tooling limitations, Kybra must dynamically upload your application's Python bytecode and the Python stdlib bytecode. This means that the Wasm module hash is not computed from all of your application's source code. Thus the Wasm module hash retrieved from the IC should not be relied upon for security assumptions.

ic.caller() in init and post_upgrade

Usually `ic.caller()` if called from `init` or `post_upgrade` is the principal of your local dfx identity. Keep in mind that in Kybra canisters `ic.caller()` if called from `init` or `post_upgrade` is the canister's own principal.

Do not use dictionary unpacking

A bug in the RustPython interpreter means that dictionary unpacking should not be used until this issue is addressed.

Reserved memory ids

`memory ids` `0` , `1` , `2` , `252` , `253` , and `254` are currently reserved for `StableBTreeMap` .

print does not work

You should use `ic.print` instead of `print` .

Kybra types

Imports

Make sure to use the `from kybra` syntax when importing types from the `kybra` module, and to not rename types with `as`:

Correct:

```
from kybra import Record

class MyRecord(Record):
    prop1: str
    prop2: int
```

Incorrect:

```
import kybra

class MyRecord(kybra.Record):
    prop1: str
    prop2: int
```

Incorrect:

```
from kybra import Record as RenamedRecord

class MyRecord(RenamedRecord):
    prop1: str
    prop2: int
```

We wish to improve this situation in the future to handle the many various ways of importing.

Treatment as keywords

You should treat Kybra types essentially as keywords, not creating types of the same name elsewhere in your codebase or in other libraries. Any types exported from this file should be treated thusly.

Reference

Call APIs

accept message

This section is a work in progress.

Examples:

- inspect_message

```python
from kybra import ic, inspect_message, void


@inspect_message
def inspect_message_() -> void:
    ic.accept_message()
```

arg data raw

This section is a work in progress.

Examples:

- ic_api

```
from kybra import blob, ic, int8, query


# returns the argument data as bytes.
@query
def arg_data_raw(arg1: blob, arg2: int8, arg3: bool, arg4: str) -> blob:
    return ic.arg_data_raw()
```

arg data raw size

This section is a work in progress.

Examples:

- ic_api

```
from kybra import blob, ic, int8, nat32, query


# returns the length of the argument data in bytes
@query
def arg_data_raw_size(arg1: blob, arg2: int8, arg3: bool, arg4: str) ->
nat32:
    return ic.arg_data_raw_size()
```

call

This section is a work in progress.

Examples:

- bitcoin
- composite_queries
- cross_canister_calls
- cycles
- ethereum_json_rpc
- func_types
- generators
- ledger_canister
- management_canister
- outgoing_http_requests
- rejections
- timers
- whoami

```
from kybra import (
    Async,
    CallResult,
    match,
    nat64,
    Principal,
    Service,
    service_update,
    update,
    Variant,
)


class TokenCanister(Service):
    @service_update
    def transfer(self, to: Principal, amount: nat64) -> nat64:
        ...


token_canister = TokenCanister(Principal.from_str("r7inp-6aaaa-aaaaa-aaabq-
cai"))


class PayoutResult(Variant, total=False):
    Ok: nat64
    Err: str


@update
def payout(to: Principal, amount: nat64) -> Async[PayoutResult]:
    result: CallResult[nat64] = yield token_canister.transfer(to, amount)

    return match(result, {"Ok": lambda ok: {"Ok": ok}, "Err": lambda err:
{"Err": err}})
```

call raw

This section is a work in progress.

Examples:

- call_raw
- outgoing_http_requests

```
from kybra import (
    Async,
    blob,
    CallResult,
    ic,
    match,
    nat64,
    Principal,
    update,
    Variant,
)


class ExecuteCallRawResult(Variant, total=False):
    Ok: str
    Err: str


@update
def execute_call_raw(
    canister_id: Principal, method: str, candid_args: str, payment: nat64
) -> Async[ExecuteCallRawResult]:
    call_result: CallResult[blob] = yield ic.call_raw(
        canister_id, method, ic.candid_encode(candid_args), payment
    )

    return match(
        call_result,
        {
            "Ok": lambda ok: {"Ok": ic.candid_decode(ok)},
            "Err": lambda err: {"Err": err},
        },
    )
```

call raw 128

This section is a work in progress.

Examples:

- call_raw

```
from kybra import (
    Async,
    blob,
    CallResult,
    ic,
    match,
    nat,
    Principal,
    update,
    Variant,
)


class ExecuteCallRaw128Result(Variant, total=False):
    Ok: str
    Err: str


@update
def execute_call_raw128(
    canister_id: Principal, method: str, candid_args: str, payment: nat
) -> Async[ExecuteCallRaw128Result]:
    call_result: CallResult[blob] = yield ic.call_raw128(
        canister_id, method, ic.candid_encode(candid_args), payment
    )

    return match(
        call_result,
        {
            "Ok": lambda ok: {"Ok": ic.candid_decode(ok)},
            "Err": lambda err: {"Err": err},
        },
    )
```

call with payment

This section is a work in progress.

Examples:

- bitcoin
- cycles
- ethereum_json_rpc
- management_canister
- outgoing_http_requests
- threshold_ecdsa

```
from kybra import Async, blob, CallResult, match, Principal, update, Variant,
void
from kybra.canisters.management import management_canister


class DefaultResult(Variant, total=False):
    Ok: bool
    Err: str


@update
def execute_install_code(
    canister_id: Principal, wasm_module: blob
) -> Async[DefaultResult]:
    call_result: CallResult[void] = yield management_canister.install_code(
        {
            "mode": {"install": None},
            "canister_id": canister_id,
            "wasm_module": wasm_module,
            "arg": bytes(),
        }
    ).with_cycles(100_000_000_000)

    return match(
        call_result, {"Ok": lambda _: {"Ok": True}, "Err": lambda err:
{"Err": err}}
    )
```

call with payment 128

This section is a work in progress.

Examples:

- cycles

```
from kybra import Async, blob, CallResult, match, Principal, update, Variant,
void
from kybra.canisters.management import management_canister


class DefaultResult(Variant, total=False):
    Ok: bool
    Err: str


@update
def execute_install_code(
    canister_id: Principal, wasm_module: blob
) -> Async[DefaultResult]:
    call_result: CallResult[void] = yield management_canister.install_code(
        {
            "mode": {"install": None},
            "canister_id": canister_id,
            "wasm_module": wasm_module,
            "arg": bytes(),
        }
    ).with_cycles128(100_000_000_000)

    return match(
        call_result, {"Ok": lambda _: {"Ok": True}, "Err": lambda err:
{"Err": err}}
    )
```

caller

This section is a work in progress.

Examples:

- ic_api
- whoami

```
from kybra import ic, Principal, query


# returns the principal of the identity that called this function
@query
def caller() -> Principal:
    return ic.caller()
```

method name

This section is a work in progress.

Examples:

- inspect_message

```
from kybra import ic, inspect_message, update, void


@inspect_message
def inspect_message_() -> void:
    ic.print("inspect_message called")

    if ic.method_name() == "accessible":
        ic.accept_message()
        return

    if ic.method_name() == "inaccessible":
        return

    raise Exception("Method " + ic.method_name() + " is not allowed")


@update
def accessible() -> bool:
    return True


@update
def inaccessible() -> bool:
    return False


@update
def also_inaccessible() -> bool:
    return False
```

msg cycles accept

This section is a work in progress.

Examples:

- cycles

```
from kybra import ic, nat64, update


@update
def receive_cycles() -> nat64:
    return ic.msg_cycles_accept(ic.msg_cycles_available() // 2)
```

msg cycles accept 128

This section is a work in progress.

Examples:

- cycles

```python
from kybra import ic, nat, update


@update
def receive_cycles128() -> nat:
    return ic.msg_cycles_accept128(ic.msg_cycles_available128() // 2)
```

msg cycles available

This section is a work in progress.

Examples:

- cycles

```
from kybra import ic, nat64, update


@update
def receive_cycles() -> nat64:
    return ic.msg_cycles_accept(ic.msg_cycles_available() // 2)
```

msg cycles available 128

This section is a work in progress.

Examples:

- cycles

```python
from kybra import ic, nat, update


@update
def receive_cycles128() -> nat:
    return ic.msg_cycles_accept128(ic.msg_cycles_available128() // 2)
```

msg cycles refunded

This section is a work in progress.

Examples:

- cycles

```
from kybra import (
    Async,
    CallResult,
    ic,
    match,
    nat64,
    Principal,
    Service,
    service_update,
    update,
    Variant,
)


class SendCyclesResult(Variant, total=False):
    Ok: nat64
    Err: str


class Cycles(Service):
    @service_update
    def receive_cycles(self) -> nat64:
        ...


cycles = Cycles(Principal.from_str("rrkah-fqaaa-aaaaa-aaaaq-cai"))


# Reports the number of cycles returned from the Cycles canister
@update
def send_cycles() -> Async[SendCyclesResult]:
    result: CallResult[nat64] = yield
cycles.receive_cycles().with_cycles(1_000_000)
    return match(
        result,
        {
            "Ok": lambda _: {"Ok": ic.msg_cycles_refunded()},
            "Err": lambda err: {"Err": err},
        },
    )
```

msg cycles refunded 128

This section is a work in progress.

Examples:

- cycles

```
from kybra import (
    Async,
    CallResult,
    ic,
    match,
    nat,
    Principal,
    Service,
    service_update,
    update,
    Variant,
)


class SendCyclesResult128(Variant, total=False):
    Ok: nat
    Err: str


class Cycles(Service):
    @service_update
    def receive_cycles128(self) -> nat:
        ...


cycles = Cycles(Principal.from_str("rrkah-fqaaa-aaaaa-aaaaq-cai"))


# Reports the number of cycles returned from the Cycles canister
@update
def send_cycles128() -> Async[SendCyclesResult128]:
    result: CallResult[nat] = yield
cycles.receive_cycles128().with_cycles128(1_000_000)

    return match(
        result,
        {
            "Ok": lambda _: {"Ok": ic.msg_cycles_refunded128()},
            "Err": lambda err: {"Err": err},
        },
    )
```

notify

This section is a work in progress.

Examples:

- cross_canister_calls
- cycles

```
from kybra import NotifyResult, Principal, Service, service_update, update,
void


class Canister2(Service):
    @service_update
    def receive_notification(self, message: str) -> void:
        ...


canister2 = Canister2(Principal.from_str("ryjl3-tyaaa-aaaaa-aaaba-cai"))


@update
def send_notification() -> NotifyResult:
    return canister2.receive_notification("This is the
notification").notify()
```

notify raw

This section is a work in progress.

Examples:

- notify_raw

```
from kybra import ic, match, Principal, RejectionCode, update, Variant


class SendNotificationResult(Variant, total=False):
    Ok: bool
    Err: RejectionCode


@update
def send_notification() -> SendNotificationResult:
    result = ic.notify_raw(
        Principal.from_str("ryjl3-tyaaa-aaaaa-aaaba-cai"),
        "receive_notification",
        ic.candid_encode("()"),
        0,
    )

    return match(
        result, {"Ok": lambda _: {"Ok": True}, "Err": lambda err: {"Err":
err}}
    )
```

notify with payment 128

This section is a work in progress.

Examples:

- cycles

```
from kybra import nat, NotifyResult, Principal, Service, service_update,
update


class Cycles(Service):
    @service_update
    def receive_cycles128(self) -> nat:
        ...


cycles = Cycles(Principal.from_str("rrkah-fqaaa-aaaaa-aaaaq-cai"))


@update
def send_cycles128_notify() -> NotifyResult:
    return cycles.receive_cycles128().with_cycles128(1_000_000).notify()
```

performance counter

This section is a work in progress.

Examples:

- ic_api

```python
from kybra import ic, nat64, query


@query
def performance_counter() -> nat64:
    return ic.performance_counter(0)
```

reject

This section is a work in progress.

Examples:

- ic_api
- manual_reply
- rejections

```
from kybra import empty, ic, Manual, query


@query
def reject(message: str) -> Manual[empty]:
    ic.reject(message)
```

reject code

This section is a work in progress.

Examples:

- rejections

```
from kybra import (
    Async,
    ic,
    Principal,
    RejectionCode,
    Service,
    service_update,
    update,
    void,
)


class Nonexistent(Service):
    @service_update
    def method(self) -> void:
        ...


nonexistent_canister = Nonexistent(Principal.from_str("rkp4c-7iaaa-aaaaa-
aaaca-cai"))


@update
def get_rejection_code_destination_invalid() -> Async[RejectionCode]:
    yield nonexistent_canister.method()
    return ic.reject_code()
```

reject message

This section is a work in progress.

Examples:

- rejections

```python
from kybra import Async, ic, Principal, Service, service_update, update, void


class SomeService(Service):
    @service_update
    def reject(self, message: str) -> void:
        ...


some_service = SomeService(Principal.from_str("rkp4c-7iaaa-aaaaa-aaaca-cai"))


@update
def get_rejection_message(message: str) -> Async[str]:
    yield some_service.reject(message)
    return ic.reject_message()
```

reply

This section is a work in progress.

Examples:

- composite_queries
- manual_reply

```
from kybra import blob, ic, Manual, update


@update
def update_blob() -> Manual[blob]:
    ic.reply(bytes([83, 117, 114, 112, 114, 105, 115, 101, 33]))
```

reply raw

This section is a work in progress.

Examples:

- manual_reply
- outgoing_http_requests

```
from kybra import blob, ic, Manual, null, query, Record, Variant


class RawReply(Record):
    int: int
    text: str
    bool: bool
    blob: blob
    variant: "Options"


class Options(Variant, total=False):
    Small: null
    Medium: null
    Large: null


@query
def reply_raw() -> Manual[RawReply]:
    ic.reply_raw(
        ic.candid_encode(
            '(record { "int" = 42; "text" = "text"; "bool" = true; "blob" =
blob "Surprise!"; "variant" = variant { Medium } })'
        )
    )
```

Candid

- blob
- bool
- empty
- float32
- float64
- func
- int
- int8
- int16
- int32
- int64
- nat
- nat8
- nat16
- nat32
- nat64
- null
- opt
- principal
- record
- reserved
- service
- text
- variant
- vec

blob

This section is a work in progress.

The Kybra type `blob` corresponds to the Candid type blob and will become a Python bytes at runtime.

Python:

```
from kybra import blob, ic, query


@query
def get_blob() -> blob:
    return bytes([68, 73, 68, 76, 0, 0])


@query
def print_blob(blob: blob) -> blob:
    ic.print(type(blob))
    return blob
```

Candid:

```
service: {
    "get_blob": () -> (blob) query;
    "print_blob": (blob) -> (blob) query;
}
```

bool

This section is a work in progress.

The Python type `bool` corresponds to the Candid type bool and will become a Python Boolean Value at runtime.

Python:

```
from kybra import ic, query


@query
def get_bool() -> bool:
    return True


@query
def print_bool(bool: bool) -> bool:
    ic.print(type(bool))
    return bool
```

Candid:

```
service: {
    "get_bool": () -> (bool) query;
    "print_bool": (bool) -> (bool) query;
}
```

empty

This section is a work in progress.

The Kybra type `empty` corresponds to the Candid type empty and has no Python value at runtime.

Python:

```
from kybra import empty, ic, query


def get_empty() -> empty:
    raise Exception("Anything you want")


# Note: It is impossible to call this function because it requires an
argument
# but there is no way to pass an "empty" value as an argument.
@query
def print_empty(empty: empty) -> empty:
    ic.print(type(empty))
    raise Exception("Anything you want")
```

Candid:

```
service: {
    "get_empty": () -> (empty) query;
    "print_empty": (empty) -> (empty) query;
}
```

float32

This section is a work in progress.

The Kybra type `float32` corresponds to the Candid type float32 and will become a Python float at runtime.

Python:

```python
import math

from kybra import float32, ic, query


@query
def get_float32() -> float32:
    return math.pi


@query
def print_float32(float32: float32) -> float32:
    ic.print(type(float32))
    return float32
```

Candid:

```
service: {
    "get_float32": () -> (float32) query;
    "print_float32": (float32) -> (float32) query;
}
```

float64

This section is a work in progress.

The Kybra type `float64` corresponds to the Candid type float64 and will become a Python float at runtime.

Python:

```python
import math

from kybra import float64, ic, query


@query
def get_float64() -> float64:
    return math.e


@query
def print_float64(float64: float64) -> float64:
    ic.print(type(float64))
    return float64
```

Candid:

```
service: {
    "get_float64": () -> (float64) query;
    "print_float64": (float64) -> (float64) query;
}
```

func

This section is a work in progress.

The Kybra type `Func` corresponds to the Candid type func and at runtime will become a Python tuple with two elements, the first being an ic-py Principal and the second being a Python str. The `ic-py Principal` represents the `principal` of the canister/service where the function exists, and the `str` represents the function's name.

Python:

```
from kybra import Func, nat64, null, Principal, query, Query, Record, Update,
Variant


class User(Record):
    id: str
    basic_func: "BasicFunc"
    complex_func: "ComplexFunc"


class Reaction(Variant, total=False):
    Good: null
    Bad: null
    BasicFunc: "BasicFunc"
    ComplexFunc: "ComplexFunc"


BasicFunc = Func(Query[[str], str])
ComplexFunc = Func(Update[[User, Reaction], nat64])


@query
def get_basic_func() -> BasicFunc:
    return (Principal.from_str("rrkah-fqaaa-aaaaa-aaaaq-cai"),
"simple_function_name")


@query
def get_complex_func() -> ComplexFunc:
    return (Principal.from_str("ryjl3-tyaaa-aaaaa-aaaba-cai"),
"complex_function_name")
```

Candid:

```
type User = record {
    "id": text;
    "basic_func": BasicFunc;
    "complex_func": ComplexFunc;
};
type Reaction = variant { "Good": null; "Bad": null; "BasicFunc": BasicFunc;
"ComplexFunc": ComplexFunc };

type BasicFunc = func (text) -> (text) query;
type ComplexFunc = func (User, Reaction) -> (nat64);

service: () -> {
    "get_basic_func": () -> (BasicFunc) query;
    "get_complex_func": () -> (ComplexFunc) query;
}
```

int

This section is a work in progress.

The Kybra type `int` corresponds to the Candid type int and will become a Python int at runtime.

Python:

```
from kybra import ic, query


@query
def get_int() -> int:
    return 170_141_183_460_469_231_731_687_303_715_884_105_727


@query
def print_int(int: int) -> int:
    ic.print(type(int))
    return int
```

Candid:

```
service: {
    "get_int": () -> (int) query;
    "print_int": (int) -> (int) query;
}
```

int8

This section is a work in progress.

The Kybra type `int8` corresponds to the Candid type int8 and will become a Python int at runtime.

Python:

```
from kybra import ic, int8, query


@query
def get_int8() -> int8:
    return 127


@query
def print_int8(int8: int8) -> int8:
    ic.print(type(int8))
    return int8
```

Candid:

```
service: {
    "get_int8": () -> (int8) query;
    "print_int8": (int8) -> (int8) query;
}
```

int16

This section is a work in progress.

The Kybra type `int16` corresponds to the Candid type int16 and will become a Python int at runtime.

Python:

```python
from kybra import ic, int16, query


@query
def get_int16() -> int16:
    return 32_767


@query
def print_int16(int16: int16) -> int16:
    ic.print(type(int16))
    return int16
```

Candid:

```
service: {
    "get_int16": () -> (int16) query;
    "print_int16": (int16) -> (int16) query;
}
```

int32

This section is a work in progress.

The Kybra type `int32` corresponds to the Candid type int32 and will become a Python int at runtime.

Python:

```
from kybra import ic, int32, query


@query
def get_int32() -> int32:
    return 2_147_483_647


@query
def print_int32(int32: int32) -> int32:
    ic.print(type(int32))
    return int32
```

Candid:

```
service: {
    "get_int32": () -> (int32) query;
    "print_int32": (int32) -> (int32) query;
}
```

int64

This section is a work in progress.

The Kybra type `int64` corresponds to the Candid type int64 and will become a Python int at runtime.

Python:

```
from kybra import ic, int64, query


@query
def get_int64() -> int64:
    return 9_223_372_036_854_775_807


@query
def print_int64(int64: int64) -> int64:
    ic.print(type(int64))
    return int64
```

Candid:

```
service: {
    "get_int64": () -> (int64) query;
    "print_int64": (int64) -> (int64) query;
}
```

nat

This section is a work in progress.

The Kybra type `nat` corresponds to the Candid type nat and will become a Python int at runtime.

Python:

```python
from kybra import ic, nat, query


@query
def get_nat() -> nat:
    return 340_282_366_920_938_463_463_374_607_431_768_211_455


@query
def print_nat(nat: nat) -> nat:
    ic.print(type(nat))
    return nat
```

Candid:

```
service: {
    "get_nat": () -> (nat) query;
    "print_nat": (nat) -> (nat) query;
}
```

nat8

This section is a work in progress.

The Kybra type `nat8` corresponds to the Candid type nat8 and will become a Python int at runtime.

Python:

```
from kybra import ic, nat8, query


@query
def get_nat8() -> nat8:
    return 255


@query
def print_nat8(nat8: nat8) -> nat8:
    ic.print(type(nat8))
    return nat8
```

Candid:

```
service: {
    "get_nat8": () -> (nat8) query;
    "print_nat8": (nat8) -> (nat8) query;
}
```

nat16

This section is a work in progress.

The Kybra type `nat16` corresponds to the Candid type nat16 and will become a Python int at runtime.

Python:

```
from kybra import ic, nat16, query


@query
def get_nat16() -> nat16:
    return 65_535


@query
def print_nat16(nat16: nat16) -> nat16:
    ic.print(type(nat16))
    return nat16
```

Candid:

```
service: {
    "get_nat16": () -> (nat16) query;
    "print_nat16": (nat16) -> (nat16) query;
}
```

nat32

This section is a work in progress.

The Kybra type `nat32` corresponds to the Candid type nat32 and will become a Python int at runtime.

Python:

```python
from kybra import ic, nat32, query


@query
def get_nat32() -> nat32:
    return 4_294_967_295


@query
def print_nat32(nat32: nat32) -> nat32:
    ic.print(type(nat32))
    return nat32
```

Candid:

```
service: {
    "get_nat32": () -> (nat32) query;
    "print_nat32": (nat32) -> (nat32) query;
}
```

nat64

This section is a work in progress.

The Kybra type `nat64` corresponds to the Candid type nat64 and will become a Python int at runtime.

Python:

```
from kybra import ic, nat64, query


@query
def get_nat64() -> nat64:
    return 18_446_744_073_709_551_615


@query
def print_nat64(nat64: nat64) -> nat64:
    ic.print(type(nat64))
    return nat64
```

Candid:

```
service: {
    "get_nat64": () -> (nat64) query;
    "print_nat64": (nat64) -> (nat64) query;
}
```

null

This section is a work in progress.

The Python type `None` and the Kybra type `null` both correspond to the Candid type null and will become the Python Null Object at runtime.

Python:

```
from kybra import ic, query


@query
def get_null() -> None:
    return None


@query
def print_null(none: None) -> None:
    ic.print(type(none))
    return none
```

Candid:

```
service: {
    "get_null": () -> (null) query;
    "print_null": (null) -> (null) query;
}
```

opt

This section is a work in progress.

The Kybra type `Opt` corresponds to the Candid type opt and will become the enclosed Python type or None at runtime.

Python:

```
from kybra import Opt, query


@query
def get_opt_some() -> Opt[bool]:
    return True


@query
def get_opt_none() -> Opt[bool]:
    return None
```

Candid:

```
service: {
    "get_opt_some": () -> (opt bool) query;
    "get_opt_none": () -> (opt bool) query;
}
```

principal

This section is a work in progress.

The Kybra type `Principal` corresponds to the Candid type principal and will become an ic-py Principal at runtime.

Python:

```
from kybra import ic, Principal, query


@query
def get_principal() -> Principal:
    return Principal.from_str("rrkah-fqaaa-aaaaa-aaaaq-cai")


@query
def print_principal(principal: Principal) -> Principal:
    ic.print(type(principal))
    return principal
```

Candid:

```
service: {
    "get_principal": () -> (principal) query;
    "print_principal": (principal) -> (principal) query;
}
```

record

This section is a work in progress.

Python classes that inherit from the Kybra type `Record` correspond to the Candid record type and will become Python TypedDicts at runtime.

Python:

```
from kybra import Record, Vec


class Post(Record):
    id: str
    author: "User"
    text: str
    thread: "Thread"


class Thread(Record):
    id: str
    author: "User"
    posts: Vec[Post]
    title: str


class User(Record):
    id: str
    posts: Vec[Post]
    thread: Vec[Thread]
    username: str
```

Candid:

```
type Post = record {
    "id": text;
    "author": User;
    "text": text;
    "thread": Thread;
};

type Thread = record {
    "id": text;
    "author": User;
    "posts": vec Post;
    "title": text;
};

type User = record {
    "id": text;
    "posts": vec Post;
    "threads": vec Thread;
    "username": text;
};
```

reserved

This section is a work in progress.

The Kybra type `reserved` corresponds to the Candid type reserved and will become the Python Null Object at runtime.

Python:

```python
from kybra import ic, query, reserved


@query
def get_reserved() -> reserved:
    return "anything"


@query
def print_reserved(reserved: reserved) -> reserved:
    ic.print(type(reserved))
    return reserved
```

Candid:

```
service: {
    "get_reserved": () -> (reserved) query;
    "print_reserved": (reserved) -> (reserved) query;
}
```

service

This section is a work in progress.

Python classes that inherit from the Kybra type `Service` correspond to the Candid service type and will become child classes capable of creating instances that can perform cross-canister calls at runtime.

Python:

```python
from kybra import (
    Async,
    CallResult,
    Principal,
    query,
    Service,
    service_query,
    service_update,
    update,
)


class SomeService(Service):
    @service_query
    def query1(self) -> bool:
        ...

    @service_update
    def update1(self) -> str:
        ...


@query
def get_service() -> SomeService:
    return SomeService(Principal.from_str("aaaaa-aa"))


@update
def call_service(service: SomeService) -> Async[str]:
    result: CallResult[str] = yield service.update1()

    if result.Err is not None:
        raise Exception(f"call to service.update1 failed with: {result.Err}")

    return result.Ok
```

Candid:

```
service { query1 : () -> (bool) query; update1 : () -> (text) }
```

text

This section is a work in progress.

The Python type `str` and the Kybra type `text` both correspond to the Candid type text and will become a Python str at runtime.

Python:

```python
from kybra import ic, query


@query
def get_string() -> str:
    return "Hello world!"


@query
def print_string(string: str) -> str:
    ic.print(type(string))
    return string
```

Candid:

```
service: {
    "get_string": () -> (text) query;
    "print_string": (text) -> (text) query;
}
```

variant

This section is a work in progress.

Python classes that inherit from the Kybra type `Variant` correspond to the Candid variant type and will become Python TypedDicts at runtime.

Python:

```python
from kybra import nat32, Variant


class ReactionType(Variant, total=False):
    Fire: None
    ThumbsUp: None
    ThumbsDown: None
    Emotion: "Emotion"
    Firework: "Firework"


class Emotion(Variant, total=False):
    Happy: None
    Sad: None


class Firework(Variant, total=False):
    Color: str
    NumStreaks: nat32
```

Candid:

```
type ReactionType = variant {
    "Fire": null;
    "ThumbsUp": null;
    "ThumbsDown": null;
    "Emotion": Emotion;
    "Firework": Firework
};

type Emotion = variant {
    "Happy": null;
    "Sad": null
};

type Firework = record {
    "Color": text;
    "NumStreaks": nat32;
};
```

vec

This section is a work in progress.

The Kybra type `Vec` corresponds to the Candid type vec and will become an array of the specified type at runtime.

Python:

```
from kybra import int32, query, Vec


@query
def get_numbers() -> Vec[int32]:
    return [0, 1, 2, 3]
```

Candid:

```
service: {
    "get_numbers": () -> (vec int32) query;
}
```

Canister APIs

- candid decode
- candid encode
- canister balance
- canister balance 128
- canister id
- data certificate
- print
- set certified data
- time
- trap

candid decode

This section is a work in progress.

Examples:

- call_raw
- candid_encoding

```
from kybra import blob, ic, query


# decodes Candid bytes to a Candid string
@query
def candid_decode(candid_encoded: blob) -> str:
    return ic.candid_decode(candid_encoded)
```

candid encode

This section is a work in progress.

Examples:

- call_raw
- candid_encoding
- manual_reply
- notify_raw
- outgoing_http_requests

```
from kybra import blob, ic, query


# encodes a Candid string to Candid bytes
@query
def candid_encode(candid_string: str) -> blob:
    return ic.candid_encode(candid_string)
```

canister balance

This section is a work in progress.

Examples:

- cycles
- ic_api

```
from kybra import ic, nat64, query


# returns the amount of cycles available in the canister
@query
def canister_balance() -> nat64:
    return ic.canister_balance()
```

canister balance 128

This section is a work in progress.

Examples:

- cycles
- ic_api

```
from kybra import ic, nat, query


# returns the amount of cycles available in the canister
@query
def canister_balance128() -> nat:
    return ic.canister_balance128()
```

canister id

This section is a work in progress.

Examples:

- ethereum_json_rpc
- ic_api
- http_counter
- outgoing_http_requests
- whoami

```python
from kybra import ic, Principal, query


# returns this canister's id
@query
def id() -> Principal:
    return ic.id()
```

data certificate

This section is a work in progress.

Examples:

- ic_api

```
from kybra import blob, ic, Opt, query


# When called from a query call, returns the data certificate authenticating
certified_data set by this canister. Returns None if not called from a query
call.
@query
def data_certificate() -> Opt[blob]:
    return ic.data_certificate()
```

print

This section is a work in progress.

Examples:

- ic_api
- null_example

You should always use `ic.print` instead of `print`.

```python
from kybra import ic, query


# prints a message through the local replica's output
@query
def print(message: str) -> bool:
    ic.print(message)

    return True
```

set certified data

This section is a work in progress.

Examples:

- ic_api

```python
from kybra import blob, ic, update, void


# sets up to 32 bytes of certified data
@update
def set_certified_data(data: blob) -> void:
    ic.set_certified_data(data)
```

time

This section is a work in progress.

Examples:

- audio_recorder
- ic_api

```
from kybra import ic, nat64, query


# returns the current timestamp
@query
def time() -> nat64:
    return ic.time()
```

trap

This section is a work in progress.

Examples:

- cross_canister_calls
- ethereum_json_rpc
- http_counter
- ic_api
- outgoing_http_requests

```
from kybra import ic, query


# traps with a message, stopping execution and discarding all state within
the call
@query
def trap(message: str) -> bool:
    ic.trap(message)

    return True
```

Canister Methods

- heartbeat
- http_request
- http_request_update
- init
- inspect message
- post upgrade
- pre upgrade
- query
- update

heartbeat

This section is a work in progress.

Examples:

- heartbeat

```
from kybra import heartbeat, ic, void


@heartbeat
def heartbeat_() -> void:
    ic.print("this runs ~1 time per second")
```

http_request

This section is a work in progress.

Examples:

- http_counter

```python
from kybra import blob, Func, nat16, Opt, query, Query, Record, Tuple,
Variant, Vec


class HttpRequest(Record):
    method: str
    url: str
    headers: Vec["Header"]
    body: blob


class HttpResponse(Record):
    status_code: nat16
    headers: Vec["Header"]
    body: blob
    streaming_strategy: Opt["StreamingStrategy"]
    upgrade: Opt[bool]


Header = Tuple[str, str]


class StreamingStrategy(Variant):
    Callback: "CallbackStrategy"


class CallbackStrategy(Record):
    callback: "Callback"
    token: "Token"


Callback = Func(Query[["Token"], "StreamingCallbackHttpResponse"])


class StreamingCallbackHttpResponse(Record):
    body: blob
    token: Opt["Token"]


class Token(Record):
    arbitrary_data: str


@query
def http_request(req: HttpRequest) -> HttpResponse:
    return {
        "status_code": 200,
        "headers": [],
        "body": bytes(),
        "streaming_strategy": None,
        "upgrade": False,
    }
```

http_request_update

This section is a work in progress.

Examples:

- http_counter

```python
from kybra import blob, Func, nat16, Opt, query, Query, Record, Tuple,
Variant, Vec


class HttpRequest(Record):
    method: str
    url: str
    headers: Vec["Header"]
    body: blob


class HttpResponse(Record):
    status_code: nat16
    headers: Vec["Header"]
    body: blob
    streaming_strategy: Opt["StreamingStrategy"]
    upgrade: Opt[bool]


Header = Tuple[str, str]


class StreamingStrategy(Variant):
    Callback: "CallbackStrategy"


class CallbackStrategy(Record):
    callback: "Callback"
    token: "Token"


Callback = Func(Query[["Token"], "StreamingCallbackHttpResponse"])


class StreamingCallbackHttpResponse(Record):
    body: blob
    token: Opt["Token"]


class Token(Record):
    arbitrary_data: str


@query
def http_request_update(req: HttpRequest) -> HttpResponse:
    return {
        "status_code": 200,
        "headers": [],
        "body": bytes(),
        "streaming_strategy": None,
        "upgrade": False,
    }
```

init

This section is a work in progress.

Examples:

- ethereum_json_rpc
- func_types
- init
- persistent-storage
- pre_and_post_upgrade
- whoami

```python
from kybra import ic, init, void


@init
def init_() -> void:
    ic.print("This runs once when the canister is first initialized")
```

inspect message

This section is a work in progress.

Examples:

- inspect_message

```python
from kybra import ic, inspect_message, update, void


@inspect_message
def inspect_message_() -> void:
    ic.print("inspect_message called")

    if ic.method_name() == "accessible":
        ic.accept_message()
        return

    if ic.method_name() == "inaccessible":
        return

    raise Exception("Method " + ic.method_name() + " is not allowed")


@update
def accessible() -> bool:
    return True


@update
def inaccessible() -> bool:
    return False


@update
def also_inaccessible() -> bool:
    return False
```

post upgrade

This section is a work in progress.

Examples:

- pre_and_post_upgrade
- whoami

```
from kybra import ic, post_upgrade, void


@post_upgrade
def post_upgrade_() -> void:
    ic.print("This runs after every canister upgrade")
```

pre upgrade

This section is a work in progress.

Examples:

- pre_and_post_upgrade

```
from kybra import ic, pre_upgrade, void


@pre_upgrade
def pre_upgrade_() -> void:
    ic.print("This runs before every canister upgrade")
```

query

This section is a work in progress.

```python
from kybra import query


@query
def simple_query() -> str:
    return "This is a query method"
```

update

This section is a work in progress.

```python
from kybra import query, update, void

message = ""


@query
def get_message() -> str:
    return message


@update
def set_message(new_message: str) -> void:
    global message
    message = new_message
```

Guard Functions

Guard functions allow you to protect your `query`, `update`, `heartbeat`, `pre_upgrade`, and `inspect_message` methods.

Guard functions return a `GuardResult` that looks like this:

```
class GuardResult(Variant, total=False):
    Ok: null
    Err: str
```

Returning `Ok` will allow the protected method to proceed with execution. Returning `Err` will reject the canister method call.

Here's a contrived example where the world is protected only 50% of the time:

```
import random

from kybra import GuardResult, update


def protect_the_world() -> GuardResult:
    if random.random() > 0.5:
        return {"Ok": None}
    else:
        return {"Err": "The world must be protected"}


@update(guard=protect_the_world)
def hello_world() -> str:
    return "Hello world!"
```

Management Canister

- bitcoin_get_balance
- bitcoin_get_current_fee_percentiles
- bitcoin_get_utxos
- bitcoin_send_transaction
- canister_status
- create_canister
- delete_canister
- deposit_cycles
- ecdsa_public_key
- http_request
- install_code
- provisional_create_canister_with_cycles
- provisional_top_up_canister
- raw_rand
- sign_with_ecdsa
- start_canister
- stop_canister
- uninstall_code
- update_settings

bitcoin_get_balance

This section is a work in progress.

Examples:

- bitcoin

```
from kybra import Async, CallResult, match, update, Variant
from kybra.canisters.management import management_canister, Satoshi

BITCOIN_API_CYCLE_COST = 100_000_000


class ExecuteGetBalanceResult(Variant, total=False):
    Ok: Satoshi
    Err: str


@update
def get_balance(address: str) -> Async[ExecuteGetBalanceResult]:
    call_result: CallResult[Satoshi] = yield
management_canister.bitcoin_get_balance(
        {"address": address, "min_confirmations": None, "network":
{"Regtest": None}}
    ).with_cycles(BITCOIN_API_CYCLE_COST)

    return match(
        call_result, {"Ok": lambda ok: {"Ok": ok}, "Err": lambda err: {"Err":
err}}
    )
```

bitcoin_get_current_fee_percentiles

This section is a work in progress.

Examples:

- bitcoin

```
from kybra import Async, CallResult, match, update, Variant, Vec
from kybra.canisters.management import management_canister,
MillisatoshiPerByte

BITCOIN_API_CYCLE_COST = 100_000_000


class GetCurrentFeePercentilesResult(Variant, total=False):
    Ok: Vec[MillisatoshiPerByte]
    Err: str


@update
def get_current_fee_percentiles() -> Async[GetCurrentFeePercentilesResult]:
    call_result: CallResult[
        Vec[MillisatoshiPerByte]
    ] = yield management_canister.bitcoin_get_current_fee_percentiles(
        {"network": {"Regtest": None}}
    ).with_cycles(
        BITCOIN_API_CYCLE_COST
    )

    return match(
        call_result, {"Ok": lambda ok: {"Ok": ok}, "Err": lambda err: {"Err":
err}}
    )
```

bitcoin_get_utxos

This section is a work in progress.

Examples:

- bitcoin

```
from kybra import Async, CallResult, match, update, Variant
from kybra.canisters.management import GetUtxosResult, management_canister

BITCOIN_API_CYCLE_COST = 100_000_000


class ExecuteGetUtxosResult(Variant, total=False):
    Ok: GetUtxosResult
    Err: str


@update
def get_utxos(address: str) -> Async[ExecuteGetUtxosResult]:
    call_result: CallResult[
        GetUtxosResult
    ] = yield management_canister.bitcoin_get_utxos(
        {"address": address, "filter": None, "network": {"Regtest": None}}
    ).with_cycles(
        BITCOIN_API_CYCLE_COST
    )

    return match(
        call_result, {"Ok": lambda ok: {"Ok": ok}, "Err": lambda err: {"Err":
err}}
    )
```

bitcoin_send_transaction

This section is a work in progress.

Examples:

- bitcoin

```
from kybra import Async, blob, CallResult, match, null, update, Variant, void
from kybra.canisters.management import management_canister

BITCOIN_BASE_TRANSACTION_COST = 5_000_000_000
BITCOIN_CYCLE_COST_PER_TRANSACTION_BYTE = 20_000_000


class SendTransactionResult(Variant, total=False):
    Ok: null
    Err: str


@update
def send_transaction(transaction: blob) -> Async[SendTransactionResult]:
    transaction_fee = (
        BITCOIN_BASE_TRANSACTION_COST
        + len(transaction) * BITCOIN_CYCLE_COST_PER_TRANSACTION_BYTE
    )

    call_result: CallResult[void] = yield
management_canister.bitcoin_send_transaction(
        {"transaction": transaction, "network": {"Regtest": None}}
    ).with_cycles(transaction_fee)

    return match(
        call_result, {"Ok": lambda ok: {"Ok": ok}, "Err": lambda err: {"Err":
err}}
    )
```

canister_status

This section is a work in progress.

Examples:

- management_canister

```
from kybra import Async, CallResult, match, update, Variant
from kybra.canisters.management import (
    CanisterStatusArgs,
    CanisterStatusResult,
    management_canister,
)


class GetCanisterStatusResult(Variant, total=False):
    Ok: CanisterStatusResult
    Err: str


@update
def get_canister_status(args: CanisterStatusArgs) ->
Async[GetCanisterStatusResult]:
    canister_status_result_call_result: CallResult[
        CanisterStatusResult
    ] = yield management_canister.canister_status({"canister_id":
args["canister_id"]})

    return match(
        canister_status_result_call_result,
        {"Ok": lambda ok: {"Ok": ok}, "Err": lambda err: {"Err": err}},
    )
```

create_canister

This section is a work in progress.

Examples:

- management_canister

```
from kybra import Async, CallResult, match, update, Variant
from kybra.canisters.management import CreateCanisterResult,
management_canister


class ExecuteCreateCanisterResult(Variant, total=False):
    Ok: CreateCanisterResult
    Err: str


@update
def execute_create_canister() -> Async[ExecuteCreateCanisterResult]:
    create_canister_result_call_result: CallResult[
        CreateCanisterResult
    ] = yield management_canister.create_canister({"settings":
None}).with_cycles(
        50_000_000_000_000
    )

    return match(
        create_canister_result_call_result,
        {"Ok": lambda ok: {"Ok": ok}, "Err": lambda err: {"Err": err}},
    )
```

delete_canister

This section is a work in progress.

Examples:

- management_canister

```
from kybra import (
    Async,
    CallResult,
    match,
    Principal,
    update,
    Variant,
    void,
)
from kybra.canisters.management import management_canister


class DefaultResult(Variant, total=False):
    Ok: bool
    Err: str


@update
def execute_delete_canister(canister_id: Principal) -> Async[DefaultResult]:
    call_result: CallResult[void] = yield
management_canister.delete_canister(
        {"canister_id": canister_id}
    )

    return match(
        call_result, {"Ok": lambda _: {"Ok": True}, "Err": lambda err:
{"Err": err}}
    )
```

deposit_cycles

This section is a work in progress.

Examples:

- management_canister

```
from kybra import Async, CallResult, match, Principal, update, Variant, void
from kybra.canisters.management import management_canister


class DefaultResult(Variant, total=False):
    Ok: bool
    Err: str


@update
def execute_deposit_cycles(canister_id: Principal) -> Async[DefaultResult]:
    call_result: CallResult[void] = yield management_canister.deposit_cycles(
        {"canister_id": canister_id}
    ).with_cycles(1_000_000)

    return match(
        call_result, {"Ok": lambda _: {"Ok": True}, "Err": lambda err:
{"Err": err}}
    )
```

ecdsa_public_key

This section is a work in progress.

Examples:

- threshold_ecdsa

```python
from kybra import Async, blob, CallResult, ic, match, Record, update, Variant
from kybra.canisters.management import management_canister,
EcdsaPublicKeyResult


class PublicKey(Record):
    public_key: blob


class PublicKeyResult(Variant, total=False):
    Ok: PublicKey
    Err: str


@update
def public_key() -> Async[PublicKeyResult]:
    caller = ic.caller().bytes
    call_result: CallResult[
        EcdsaPublicKeyResult
    ] = yield management_canister.ecdsa_public_key(
        {
            "canister_id": None,
            "derivation_path": [caller],
            "key_id": {"curve": {"secp256k1": None}, "name": "dfx_test_key"},
        }
    )

    return match(
        call_result, {"Ok": lambda ok: {"Ok": ok}, "Err": lambda err: {"Err":
err}}
    )
```

http_request

This section is a work in progress.

Examples:

- ethereum_json_rpc
- outgoing_http_requests

```
from kybra import Async, CallResult, match, ic, query, update
from kybra.canisters.management import (
    HttpResponse,
    HttpTransformArgs,
    management_canister,
)


@update
def xkcd() -> Async[HttpResponse]:
    max_response_bytes = 1_000

    # TODO this is just a heuristic for cost, might change when the feature
is officially released: https://forum.dfinity.org/t/enable-canisters-to-make-
http-s-requests/9670/130
    cycle_cost_base = 400_000_000
    cycle_cost_per_byte = 300_000  # TODO not sure on this exact cost
    cycle_cost_total = cycle_cost_base + cycle_cost_per_byte *
max_response_bytes

    http_result: CallResult[HttpResponse] = yield
management_canister.http_request(
        {
            "url": "https://xkcd.com/642/info.0.json",
            "max_response_bytes": max_response_bytes,
            "method": {"get": None},
            "headers": [],
            "body": None,
            "transform": {"function": (ic.id(), "xkcd_transform"), "context":
bytes()},
        }
    ).with_cycles(cycle_cost_total)

    return match(http_result, {"Ok": lambda ok: ok, "Err": lambda err:
ic.trap(err)})


@query
def xkcd_transform(args: HttpTransformArgs) -> HttpResponse:
    http_response = args["response"]

    http_response["headers"] = []

    return http_response
```

install_code

This section is a work in progress.

Examples:

- management_canister

```
from kybra import (
    Async,
    blob,
    CallResult,
    match,
    Principal,
    update,
    Variant,
    void,
)
from kybra.canisters.management import management_canister


class DefaultResult(Variant, total=False):
    Ok: bool
    Err: str


@update
def execute_install_code(
    canister_id: Principal, wasm_module: blob
) -> Async[DefaultResult]:
    call_result: CallResult[void] = yield management_canister.install_code(
        {
            "mode": {"install": None},
            "canister_id": canister_id,
            "wasm_module": wasm_module,
            "arg": bytes(),
        }
    ).with_cycles(100_000_000_000)

    return match(
        call_result, {"Ok": lambda _: {"Ok": True}, "Err": lambda err:
{"Err": err}}
    )
```

provisional_create_canister_with_cycles

This section is a work in progress.

Examples:

- management_canister

```
from kybra import Async, CallResult, match, update, Variant
from kybra.canisters.management import (
    CreateCanisterResult,
    management_canister,
    ProvisionalCreateCanisterWithCyclesResult,
)


class ExecuteProvisionalCreateCanisterWithCyclesResult(Variant, total=False):
    Ok: CreateCanisterResult
    Err: str


@update
def provisional_create_canister_with_cycles() -> (
    Async[ExecuteProvisionalCreateCanisterWithCyclesResult]
):
    call_result: CallResult[
        ProvisionalCreateCanisterWithCyclesResult
    ] = yield management_canister.provisional_create_canister_with_cycles(
        {"amount": None, "settings": None}
    )

    return match(
        call_result, {"Ok": lambda ok: {"Ok": ok}, "Err": lambda err: {"Err":
err}}
    )
```

provisional_top_up_canister

This section is a work in progress.

Examples:

- management_canister

```
from kybra import (
    Async,
    CallResult,
    match,
    nat,
    Principal,
    update,
    Variant,
    void,
)
from kybra.canisters.management import management_canister


class DefaultResult(Variant, total=False):
    Ok: bool
    Err: str


@update
def provisional_top_up_canister(
    canister_id: Principal, amount: nat
) -> Async[DefaultResult]:
    call_result: CallResult[
        void
    ] = yield management_canister.provisional_top_up_canister(
        {"canister_id": canister_id, "amount": amount}
    )

    return match(
        call_result, {"Ok": lambda _: {"Ok": True}, "Err": lambda err:
{"Err": err}}
    )
```

raw_rand

This section is a work in progress.

Examples:

- generators
- heartbeat
- management_canister
- timers

```
from kybra import Async, blob, CallResult, match, update
from kybra.canisters.management import management_canister


@update
def get_randomness_directly() -> Async[blob]:
    randomness_result: CallResult[blob] = yield
management_canister.raw_rand()

    return match(randomness_result, {"Ok": lambda ok: ok, "Err": lambda err:
err})
```

sign_with_ecdsa

This section is a work in progress.

Examples:

- threshold_ecdsa

```
from kybra import Async, blob, CallResult, ic, match, Record, update, Variant
from kybra.canisters.management import (
    management_canister,
    SignWithEcdsaResult,
)


class Signature(Record):
    signature: blob


class SignResult(Variant, total=False):
    Ok: Signature
    Err: str


@update
def sign(message_hash: blob) -> Async[SignResult]:
    if len(message_hash) != 32:
        ic.trap("message_hash must be 32 bytes")

    caller = ic.caller().bytes

    call_result: CallResult[
        SignWithEcdsaResult
    ] = yield management_canister.sign_with_ecdsa(
        {
            "message_hash": message_hash,
            "derivation_path": [caller],
            "key_id": {"curve": {"secp256k1": None}, "name": "dfx_test_key"},
        }
    ).with_cycles(
        10_000_000_000
    )

    return match(
        call_result, {"Ok": lambda ok: {"Ok": ok}, "Err": lambda err: {"Err":
err}}
    )
```

start_canister

This section is a work in progress.

Examples:

- management_canister

```
from kybra import (
    Async,
    CallResult,
    match,
    Principal,
    update,
    Variant,
    void,
)
from kybra.canisters.management import management_canister


class DefaultResult(Variant, total=False):
    Ok: bool
    Err: str


@update
def execute_start_canister(canister_id: Principal) -> Async[DefaultResult]:
    call_result: CallResult[void] = yield management_canister.start_canister(
        {"canister_id": canister_id}
    )

    return match(
        call_result, {"Ok": lambda _: {"Ok": True}, "Err": lambda err:
{"Err": err}}
    )
```

stop_canister

This section is a work in progress.

Examples:

- management_canister

```
from kybra import (
    Async,
    CallResult,
    match,
    Principal,
    update,
    Variant,
    void,
)
from kybra.canisters.management import management_canister


class DefaultResult(Variant, total=False):
    Ok: bool
    Err: str


@update
def execute_stop_canister(canister_id: Principal) -> Async[DefaultResult]:
    call_result: CallResult[void] = yield management_canister.stop_canister(
        {"canister_id": canister_id}
    )

    return match(
        call_result, {"Ok": lambda _: {"Ok": True}, "Err": lambda err:
{"Err": err}}
    )
```

uninstall_code

This section is a work in progress.

Examples:

- management_canister

```
from kybra import Async, CallResult, match, Principal, update, Variant, void
from kybra.canisters.management import management_canister


class DefaultResult(Variant, total=False):
    Ok: bool
    Err: str


@update
def execute_uninstall_code(canister_id: Principal) -> Async[DefaultResult]:
    call_result: CallResult[void] = yield management_canister.uninstall_code(
        {"canister_id": canister_id}
    )

    return match(
        call_result, {"Ok": lambda _: {"Ok": True}, "Err": lambda err:
{"Err": err}}
    )
```

update_settings

This section is a work in progress.

Examples:

- management_canister

```
from kybra import Async, CallResult, match, Principal, update, Variant, void
from kybra.canisters.management import management_canister


class DefaultResult(Variant, total=False):
    Ok: bool
    Err: str


@update
def execute_update_settings(canister_id: Principal) -> Async[DefaultResult]:
    call_result: CallResult[void] = yield
management_canister.update_settings(
        {
            "canister_id": canister_id,
            "settings": {
                "controllers": None,
                "compute_allocation": 1,
                "memory_allocation": 3_000_000,
                "freezing_threshold": 2_000_000,
            },
        }
    )

    return match(
        call_result, {"Ok": lambda _: {"Ok": True}, "Err": lambda err:
{"Err": err}}
    )
```

Stable Memory

- stable structures
- stable bytes
- stable grow
- stable read
- stable size
- stable write
- stable64 grow
- stable64 read
- stable64 size
- stable64 write

stable structures

This section is a work in progress.

Examples:

- audio_recorder
- ethereum_json_rpc
- func_types
- http_counter
- persistent-storage
- pre_and_post_upgrade
- stable_structures

```
from kybra import (
    Alias,
    nat64,
    nat8,
    Opt,
    query,
    StableBTreeMap,
    Tuple,
    update,
    Vec,
)

Key = Alias[nat8]
Value = Alias[str]


map = StableBTreeMap[Key, Value](memory_id=3, max_key_size=100,
max_value_size=1_000)


@query
def contains_key(key: Key) -> bool:
    return map.contains_key(key)


@query
def get(key: Key) -> Opt[Value]:
    return map.get(key)


@update
def insert(key: Key, value: Value) -> Opt[Value]:
    return map.insert(key, value)


@query
def is_empty() -> bool:
    return map.is_empty()


@query
def items() -> Vec[Tuple[Key, Value]]:
    return map.items()


@query
def keys() -> Vec[Key]:
    return map.keys()


@query
def len() -> nat64:
    return map.len()


@update
```

```
def remove(key: Key) -> Opt[Value]:
    return map.remove(key)


@query
def values() -> Vec[Value]:
    return map.values()
```

stable bytes

This section is a work in progress.

Examples:

- stable_memory

```
from kybra import blob, ic, query


@query
def stable_bytes() -> blob:
    return ic.stable_bytes()
```

stable grow

This section is a work in progress.

Examples:

- stable_memory

```
from kybra import ic, nat32, StableGrowResult, update


@update
def stable_grow(new_pages: nat32) -> StableGrowResult:
    return ic.stable_grow(new_pages)
```

stable read

This section is a work in progress.

Examples:

- stable_memory

```python
from kybra import blob, ic, nat32, query


@query
def stable_read(offset: nat32, length: nat32) -> blob:
    return ic.stable_read(offset, length)
```

stable size

This section is a work in progress.

Examples:

- stable_memory

```
from kybra import ic, nat32, query


@query
def stable_size() -> nat32:
    return ic.stable_size()
```

stable write

This section is a work in progress.

Examples:

- stable_memory

```
from kybra import blob, ic, nat32, update, void


@update
def stable_write(offset: nat32, buf: blob) -> void:
    ic.stable_write(offset, buf)
```

stable64 grow

This section is a work in progress.

Examples:

- stable_memory

```
from kybra import ic, nat64, Stable64GrowResult, update


@update
def stable_grow(new_pages: nat64) -> Stable64GrowResult:
    return ic.stable_grow(new_pages)
```

stable64 read

This section is a work in progress.

Examples:

- stable_memory

```python
from kybra import blob, ic, nat64, query


@query
def stable64_read(offset: nat64, length: nat64) -> blob:
    return ic.stable64_read(offset, length)
```

stable64 size

This section is a work in progress.

Examples:

- stable_memory

```
from kybra import ic, nat64, query


@query
def stable64_size() -> nat64:
    return ic.stable64_size()
```

stable64 write

This section is a work in progress.

Examples:

- stable_memory

```python
from kybra import blob, ic, nat64, update, void


@update
def stable64_write(offset: nat64, buf: blob) -> void:
    ic.stable64_write(offset, buf)
```

Timers

- clear timer
- set timer
- set timer interval

clear timer

This section is a work in progress.

Examples:

- timers

```
from kybra import ic, TimerId, update, void


@update
def clear_timer(timer_id: TimerId) -> void:
    ic.clear_timer(timer_id)
    ic.print(f"timer {timer_id} cancelled")
```

set timer

This section is a work in progress.

Examples:

- timers

```
from kybra import (
    Duration,
    ic,
    TimerId,
    update,
)


@update
def set_timer(delay: Duration) -> TimerId:
    return ic.set_timer(delay, timer_callback)


def timer_callback():
    ic.print("timer_callback")
```

set timer interval

This section is a work in progress.

Examples:

- timers

```python
from kybra import (
    Duration,
    ic,
    TimerId,
    update,
)

counter = 0


@update
def set_timer_interval(interval: Duration) -> TimerId:
    return ic.set_timer_interval(interval, timer_callback)


def timer_callback():
    global counter
    counter += 1

    ic.print(f"timer_callback: {counter}")
```