# community-and-support.md:

Community and support

Join our community

- Join our Discord
- Telegram groups
  - Subscribe to Nillion Official Announcements to stay up to date with the latest Nillion news
  - Join MPC 👀, a broader community group for discussing MPC developments, papers, and implementation examples.
- Visit NillHub, our community discussion forum. This is a place where all Nills from around the globe can share, learn, and create together, and for anyone whose native language isn’t English, there are sub-categories for languages with their own communities too.

Developer questions and technical discussions

Nillion's Github Discussions forum is our central developer hub for technical discussions related to Nillion. Builders can use Github discussions to share ideas, ask questions, showcase their projects, and stay updated on the latest news and announcements from the Nillion team.

Improve the docs

Got a suggestion for improving the docs? Let us know by creating a Github Issue.

How to open a support ticket

1. Join our Discord
2. Visit the #support-ticket Discord channel
3. Create a ticket (General support or Partnership request)

# compute-js.md:

```
import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
```

Compute

Perform computation using secrets stored in the network.

Get a Quote to Compute

```
<Tabs>

<TabItem value="getquote" label="Get quote to compute" default>
ts reference showGithubLink
https://github.com/NillionNetwork/cra-nillion/blob/main/src/nillion/components/ComputeForm.tsx#L53-L60

</TabItem>

<TabItem value="getQuote" label="getQuote helper">
ts reference showGithubLink
https://github.com/NillionNetwork/cra-nillion/blob/main/src/nillion/helpers/getQuote.ts

</TabItem>
```

</Tabs>

Pay to Compute and get Payment Receipt

<Tabs>

<TabItem value="receipt" label="Payment receipt" default>
ts reference showGithubLink
https://github.com/NillionNetwork/cra-nillion/blob/main/src/nillion/components/ComputeForm.tsx#L74-L81

</TabItem>

<TabItem value="helpers" label="helper functions">
ts reference showGithubLink
https://github.com/NillionNetwork/cra-nillion/blob/main/src/nillion/helpers/nillion.ts#L24-L71

</TabItem>

</Tabs>

Compute

<Tabs>
<TabItem value="compute" label="compute" default>
ts reference showGithubLink
https://github.com/NillionNetwork/cra-nillion/blob/main/src/nillion/components/ComputeForm.tsx#L85-L94

</TabItem>

<TabItem value="helpers" label="computeProgram helper">
ts reference showGithubLink
https://github.com/NillionNetwork/cra-nillion/blob/main/src/nillion/helpers/compute.ts

</TabItem>

</Tabs>

# compute.md:

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import DocCardList from '@theme/DocCardList';

Compute

Perform single or multi party computation using secrets stored in the network.

<DocCardList/>

Single Party Compute

Single party compute involves only one Party that provides inputs and receives outputs of a program. Single party compute examples are available in the Python Starter Repo coreconceptsinglepartycompute folder.

Example: additionsimple.py

The additionsimple example is a single party compute example that adds two secret integers. In the client code, the first secret integer is stored in the network ahead of time, and the second secret integer is provided at computation time.

```
<Tabs>
  <TabItem value="client" label="Client code" default>
```

python reference showGithubLink
https://github.com/NillionNetwork/python-examples/tree/main/examplesandtutorials/coreconceptsinglepartycompute/additionsimple.py#L14-L131

```
  </TabItem>
  <TabItem value="readme" label="Nada program" default>
```

python reference showGithubLink
https://github.com/NillionNetwork/python-examples/blob/main/examplesandtutorials/nadaprograms/src/additionsimple.py

```
  </TabItem>
</Tabs>
```

Multi Party Compute

Multi party compute involves more than one Party. These Parties collaborate to provide secret inputs and one Party receives outputs of the program.

The coreconceptmultipartycompute folder has a 3-step multi party compute example involving multiple parties providing secret inputs for computation of a program. The first party stores a secret, then N other parties store permissioned secrets giving the first party compute access. The first party computes with all secrets.

```
<Tabs>
  <TabItem value="readme" label="README" default>
```

python reference showGithubLink
https://github.com/NillionNetwork/python-examples/tree/main/examplesandtutorials/coreconceptmultipartycompute/README.md

```
  </TabItem>
  <TabItem value="config" label="Config file" default>
```

python reference showGithubLink
https://github.com/NillionNetwork/python-examples/tree/main/examplesandtutorials/coreconceptmultipartycompute/config.py

```
  </TabItem>
  <TabItem value="apple" label="Step 1" default>
    Step 1: 1st Party Stores a Secret
```

python reference showGithubLink
https://github.com/NillionNetwork/python-examples/tree/main/examplesandtutorials/coreconceptmultipartycompute/01storesecretparty1.py#L19-L100

  </TabItem>
  <TabItem value="orange" label="Step 2">
    Step 2: N other parties store a permissioned secret

python reference showGithubLink
https://github.com/NillionNetwork/python-examples/tree/main/examplesandtutorials/coreconceptmultipartycompute/02storesecretpartyn.py#L36-L108

  </TabItem>
  <TabItem value="banana" label="Step 3">
    Step 3: The 1st Party computes with all secrets

python reference showGithubLink
https://github.com/NillionNetwork/python-examples/tree/main/examplesandtutorials/coreconceptmultipartycompute/03multipartycompute.py#L43-L100

  </TabItem>
</Tabs>

# concepts.md:

Key Concepts

Users

A Nillion Network user has a Ed25519 key pair consisting of a public and private key.

User Key

The userkey is the user's private key. The user key should never be shared publicly, as it unlocks access and permissions to secrets stored on the network.

User ID

The userid is derived from the user's public key, and is the public user identifier. Other parties can grant a user permission to a secret based on their user id.&#x20;

Programs

A Nada program is also a piece of software that is written in Nada language, then compiled to generate a mid-level intermediate (MIR) representation of the program that can be stored in the Nillion Network for future use. Programs are characterized by a set of inputs, the computation logic, and a set of outputs. Programs are reusable, and computation is invoked by a Nillion Client. At computation time, the client specifies inputs, which can be any combination of Secrets already stored in the network, secrets provided at compute time, and public variables.

Program ID

The programid is the identifier for a program stored in the Nillion Network. The convention for programid is the {userid}/{programname} where the userid corresponds to the user that stored the program in the network, and the programname is the program name the storer set when storing the program.

Party

A Party is a named entity involved in Nada program computation. Parties can supply inputs to calculations, reveal outputs of calculations, or both.

Party ID

The partyid, sometimes called peerid, is the public identifier for a Party.

Inputs

An Input is named external data provided by a specific Party for Nada program computation. Inputs can be secret or public.

Outputs

An Output is the named result of a Nada program revealed to a specific Party after computation.

Secrets

A Nillion Secret, identified in the network by a storeid, consists of one or more named secret values. Secrets can be stored in the network or provided as inputs to programs at compute time.

Store ID

The storeid is the identifier for a set of one or more named secret values stored in the Nillion Network. This storeid is returned by the network as a result of storing a secret.

Secret Name

The secretname is a user given name for a single secret value.

Permissions

Permissions are access control mechanisms attached to a secret stored in the network

Default Permissions

Any userid given "default permissions" at store time through defaultforuser(userid) will have permission to retrieve and update the permissions of a secret.

Secret Permissions

A userid can be given retrieve, update, delete, and/or compute permissions on a secret.

- retrieve / read a secret: addretrievepermissions(userid)
- update a secret: addupdatepermissions(userid)
- delete a secret: adddeletepermissions(userid)
- compute on a secret: addcomputepermissions({userid: {programid}})

Clusters

A cluster is a subset of the compute nodes that exist in the Nillion Network.

Cluster ID

The clusterid is the identifier for a cluster of nodes in the Nillion Network.

Nodes

Nodes in the Nillion Network have public and private key pairs.

Node Key

The nodekey is the node's private key, which it keeps a secret.

Peer ID

The peerid is a libp2p concept. Each node is a peer in the overall peer-to-peer Nillion Network. The peerid serves as a unique identifier for each peer or node, and is a verifiable link between a peer and its public cryptographic key.

Bootnodes

Bootnodes are nodes designated as entry points of the Nillion Network. All the nodes require a list of bootnodes to be able to join the network. After a node dials the configured bootnodes and connects, the bootnodes introduce them to the rest of the network allowing the peer discovery process to start.

Dealer nodes

A dealer node is an SDK-based node that sends tasks to the Nillion Network.

Result nodes

A result node is an SDK-based node that receives the results of computations run on the Nillion Network.

Compute nodes

Compute nodes can perform all the functions of the Nillion Network including computation and storage.

Relay servers

Because dealer and result nodes can run anywhere (web browsers, other platforms), compute nodes are configured as relay servers with libp2p's Circuit Relay protocol. Dealers and result nodes can establish relay circuits with relay servers in order to operate in the network the same as the rest of the nodes.

# data-wars.md:

Data Wars: Community Ranks

This is not the allowlist you are looking you're looking for. Join the multi-week Nillion bootcamp to be fully onboarded into the Nil Army. Every week you will have the potential to be whitelisted for the upcoming Nillion Blindfolded PFP NFT release.

!An image from the static

A rebellion is brewing in the shadows. The Data War draft is now OPEN.

Nillion is drafting supporters, contributors, builders and creative thinkers into its army.

Rewards and glory await those who excel.

Time to select your class:

Connect

The most important aspect of being a fighter in the Nillion army is spreading the word creatively by communicating and connecting with other data fighters.

We have two main groups for chatting - the Discord and the Telegram.

At Nillion, everything needs to be high quality, including our communications. Picture this: every conversation you have within the community serves as a welcoming handshake for new members. So if you’re having engaging, interesting, high quality conversations, we will attract more interesting people to join our cause and get involved.

If we are going to usher in a true data renaissance, we need people like you to champion Nillion’s message:

Private data for all.

Connect classes

If you engage in meaningful conversations that enhance the community, you'll earn the Warrior class. However, farming this role will get you banned.

If you have special interests or post in channels about trading, tech, NFTs, DeFi, AI, Privacy, Data, DePin you will be given the Sage class.

If you do any of these really well and the community recommends you, then the “Berserker” role will be granted.

If you’re an integral member of the community, then you’ll be awarded the highest rank: The Marshal class. This rank grants access to the exclusive Marshall channel.

Remember! Community isn't just in Discord and Telegram

Whenever posts come from the Nillion main account or team members like Charlie, Alex, Miguel, or 0xCrayon be sure to reply with a thought, a joke, a meme, or anything that makes sense and is engaging.

Scribe

Want to create and contribute in a more substantial way? Then you can write about Nillion and our vision for the future.

If you are going to write about Nillion and our plans to win the data war (blind computation, privacy, data ownership) then do it with personality and conviction.

We refuse to blend in with the crowd of mundane, overlooked deep tech projects. That’s not the Nillion ethos. That’s not how you leave a mark.

The rule applies for memes and art as well. We cherish originality. Craft your unique style and artwork inspired by Nillion, something the community will embrace and contribute to.

Don’t create based on what you assume we want to see - create what resonates with you.

If you are a photographer then take photos, if you’re a musician then create music, if you’re a writer then create lore. If you don’t currently do anything then learn something new. Spend some time with the rest of the community to learn a new skill and use Nilions ideas, visions, and beliefs as cues to create something amazing and impactful.

Scribe Classes

If you are full of creative ideas and love to create memes, you’ll earn the class of Master of Memes Class

We also have some specific roles for those who are more creative.

Designer: A role for people who create great art/visuals

Visionary: A role for people who create great videos/gifs/animated content

Build

Are you a builder, someone who is building the technology that is going to propel us into the data renaissance?

Builder Classes

If you like to participate in hackathons or developer competitions around and build blind applications then you will earn the Hacker class.

If you want to use, build with, or create content around the Nillion SDK then you can earn the Engineer class.

If you are someone who likes to dive into the code and make improvements/find bugs the Tinker class might be for you.

Raid

Share on Twitter or to your Telegram groups or wherever you are most active. Add some personality and describe what you’re sharing and where people can find more.

Share your posts with the community and we will all jump in and engage.

Remember classes are earned not given. We are always watching our army, and no good contribution will go unnoticed.

Our soldiers are outfitted with blue blindfolds, armed with digital weapons to combat the powers that be.

Every good input will increase your rank within the Data War.

Good luck, comrade.

# first-steps.md:

Your first steps with Nillion

👋 Hey, welcome to Nillion.

This page will help you take your first steps (30 mins) as a Nillion developer. Once you have completed them, fill out the form for your chance to claim your $20 prize - each week we will review the submissions and pick the best to recieve a prize 🎉

1. Star & fork either the Python or JavaScript quickstart repos (bonus points for both)

   📥 Add the topic nillion-nada to this repo so we can find it easily

2. Follow at least one of the quickstarts: Python Quickstart or JavaScript Blind App Quickstart

   📥 Make sure you enable telemetry as you install the Nillion SDK

3. Once you have completed the quickstart, add at least one new nada program to your repo (bonus points for creativity, but make sure it compiles and runs)

   - Python quickstart: add your nada program in your repo's programs folder
   - JavaScript quickstart: add your program in your repo's public/programs folder

4. Join our community:

   - Join our Discord server
   - Follow us on Twitter

5. Fill out this form to claim your prize 🎉

# glossary.md:

```
---
layout:
  title:
    visible: true
  description:
    visible: true
  tableOfContents:
    visible: true
  outline:
    visible: true
  pagination:
    visible: true
---
```

Glossary

Information-theoretic security (ITS)

Information-theoretic security is a measure of security that doesn't depend on computational hardness assumptions. ITS guarantees security even against adversaries with unlimited computational power. This type of security is achieved by ensuring that the information required to break the encryption is not present in the cipher text. A classic example of ITS is a one-time pad cipher, which is provably unbreakable as long as the key is truly random, never reused, and kept secret.

Linear Secret Sharing Scheme (LSSS)

A Linear Secret Sharing Scheme is a cryptography method where a secret is divided into multiple parts, known as shares. These shares are distributed among participants in such a way that only specific groups of shares can reconstruct the secret. The key property of LSSS is linearity, meaning that any linear combination of valid shares forms another valid share. This allows for flexibility in constructing access structures, determining which sets of participants can together reconstruct the secret.

Masked factors

Masked factors are the protective factors in the Sum of Products. Masked factors are the one-time mask raised to the power of the masked exponent multiplied by the factor.

One-time mask

The one-time mask is the multiplicative mask that protects a factor.&#x20;

Privacy-enhancing technologies (PETs)

Privacy-enhancing technologies (PETs) are technologies that embody fundamental data protection principles by minimizing personal data use, maximizing data security, and empowering individuals.

Share

The share is the result that a node obtains by multiplying all the masked factors, and adding up all the products of those masked factors using the sum-of-products expression shape.

Sum of Products (SoP)

Sum of Products is a mathematical expression of addition of several numbers that have been multiplied.

```
python
Example of a Sum of Products
(a  b) + (c  d)
```

Any computation can be transformed into a sum of products:

```
python
Example of a polynomial computation
x^2 + 2xy + y^2

Transformation to a sum of products
(x  x) + (2  x  y) + (y  y)
```

Sum of Products (SoP) - Factors and Terms <a href="#factors-and-terms" id="factors-and-terms"></a>

- Terms: In SoP, a term is a product of factors that are multiplied together. In the example SoP, the terms are (a \ b) and (c \ d). Each term represents a multiplication operation between its factors.
- Factor: Factors are the individual elements that are multiplied together within a term. In the example SoP, a and b are factors in the (a\b) term. c and d are factors in the (c\d) term. They are the basic building blocks of a term.

# guide-testnet-connect.md:

```
import IframeVideo from '@site/src/components/IframeVideo/index';
import SupportedWallets from './\testnetsupportedwallets.mdx';
```

Creating a Nillion Wallet

Create a Nillion wallet to connect to the Nillion Testnet and access your assets.

Supported Wallets

<SupportedWallets/>

How to create a Nillion wallet and connect to the Nillion Testnet

This guide walks you through downloading a wallet, setting up your wallet, and adding the Nillion Testnet.

1. Download Keplr

Visit the Keplr download page and select your browser to download the Keplr browser extension. We recommend Chrome.

!Download Keplr

2. Install the Keplr browser extension

Click "Add to Chrome" to install the Keplr Chrome extension.

!Add extension

3. Create a new wallet

After Keplr is installed, it will pop open in a new tab. Click "Create a new wallet."

!Create wallet

4. Create a recovery phrase

Create a wallet using a recovery phrase. Click "Create new recovery phrase."

!Create a wallet with a new recovery phrase

5. Secure your recovery phrase

Follow the steps to show your recovery phrase. Make sure to store your secret recovery phrase securely.
!Secure phrase

6. Set a wallet password

Verify your recovery phrase by filling in ordered words. Then name your wallet and set a password. You'll use this password to log into Keplr on this device. To import this wallet for use on another device, you'll need the recovery phrase.
!Verify

7. Add Nillion Testnet

    1. Toggle open the "Extensions" button in your browser and pin the Keplr extension so that the browser has access to it.

    2. Open the Keplr Chains page and search for "Nillion." Click "Add to Keplr."

    3. "Approve" adding the Nillion Testnet within Keplr.

    4. In the Keplr extension, click the hamburger menu and click "Manage Chain Visibility." This opens the Select Chains page in a new tab. Search for "NIL" and click the checkmark to make the Nillion Testnet visible from within Keplr. Click "Save."

    5. Verify setup by toggling Keplr back open. Scroll down to the bottom of the "Home" tab, where you should see a new NIL token balance of 0 NIL.

8. Find your new Nillion wallet address

    1. From your Keplr extension, click the "Copy address" button.

    2. Now you'll see addresses for all enabled chains. Filter for "NIL"

    3. Click to copy your Nillion wallet address to your clipboard. Your Nillion wallet address is safe to share with others; they'll need your wallet address to send you NIL.

<IframeVideo
videoSrc="https://www.loom.com/embed/3b243bee264d4ca992381ef131e5a625?
sid=17c8f87a-a468-41e3-88f1-7ca287063d29"/>

<br/>
Nice! You've set up your Nillion wallet and have a Nillion wallet address. Next, use the Faucet to get some Testnet NIL.

# guide-testnet-faucet.md:

import IframeVideo from '@site/src/components/IframeVideo/index';

Using the Faucet

The Nillion Testnet Faucet distributes Nillion Testnet NIL tokens. You can use the faucet to request tokens once every 24 hours.

How to use the Nillion Testnet Faucet

This guide walks you through how to get NIL from the Nillion Testnet Faucet.

0. Set up a wallet

Follow the Creating a Nillion Wallet guide to set up a connected wallet.

1. Go to the Nillion Faucet page.

Click the "Start" button.

!faucet

2. Add your Nillion Testnet wallet address.

    1. Open the Keplr extension and click "Copy address."

    2. Search for "Nillion Testnet"

    3. Click the copy icon next to "Nillion Testnet" to copy your Nillion Testnet wallet address.

    4. Paste your Nillion Testnet wallet address into the faucet.

    5. Click "Continue."

<IframeVideo
videoSrc="https://www.loom.com/embed/d47a393e87544095a4bbf5531aac79f2?
sid=3d0b8ee1-7c74-4c71-82f8-41ecb463e838"/>

3. Complete the verification challenge.

Complete the verification challenge and then click "Continue." The faucet will send testnet NIL to your Nillion Testnet wallet address.

4. Check your wallet for a NIL balance

Open your wallet to see the NIL in your Nillion Testnet wallet.

<IframeVideo
videoSrc="https://www.loom.com/embed/93703c126ae74c8a9ff55e5d33063395?
sid=aa08c50f-0aff-4d4f-9eca-70cb774736b4"/>

<br/>

Great work! Your Nillion Testnet wallet is funded with some Testnet NIL. Next make a transaction on the Nillion Testnet by sending some Testnet NIL to a friend's Nillion Testnet wallet.

# guide-testnet-tx.md:

import IframeVideo from '@site/src/components/IframeVideo/index';
import BlockExplorers from './\testnetblockexplorers.mdx';

Sending NIL Tokens

Send a transaction on the Nillion Testnet and find the onchain record of the transaction on a block explorer.

How to send NIL

This guide walks you through how to send NIL testnet tokens from one Nillion Testnet wallet to another Nillion Testnet wallet and find the transaction on a block explorer.

0. Set up a wallet and get NIL

Follow the Creating a Nillion Wallet guide to set up a connected wallet.

Follow the Using the Faucet guide to request NIL testnet tokens.

1. Open Keplr and check your NIL balance

    1. Open Keplr and find the "Search for asset or chain" search bar on the Home tab.

    2. Type in "NIL" to filter for your available balance of NIL.

    3. Click "NIL" to open the NIL on Nillion Testnet view.

2. Send NIL to another Nillion wallet

    1. Click "Send"

    2. Under "Wallet address" paste in a valid Nillion wallet address - this address should start with nillion1

    3. Under "Amount" type in the amount of NIL to send or click "Max" to send all your NIL minus the transaction fee (Tx Fee) shown at the bottom of Keplr.

    4. Under "Memo", optionally add a note to be sent with the transaction.

    5. Click "Next."

    6. Confirm the transaction by reviewing details to make sure the recipient's wallet address, amount, and memo are correct.

    7. Click "Approve" to send the transaction, and wait for a success message to appear.

    <IframeVideo videoSrc="https://www.loom.com/embed/bb37a3c363424e4385636a4b9cbf779e?sid=32c65c3e-247f-441a-af95-b879c4dbff1c"/>

3. Look for the transaction on a Nillion block explorer

There are multiple Nillion Testnet block explorers you can use to see your transaction.

<BlockExplorers/>

Use an explorer to find your transaction:

1. Open a Nillion Testnet block explorer and paste your Nillion wallet address into the search bar.

2. Find "Transactions" and click on the latest transaction.

3. Look at the transaction details to see the onchain record of the NIL you just sent.

<IframeVideo
videoSrc="https://www.loom.com/embed/6f9023f29ad547f4b3a4f92bc852c11c?sid=cd8061b7-6ccf-4523-b0c9-094193e084d6"/>

<br/>

Well done! You just sent your first transaction on the Nillion Testnet!

# hacker-house-goa-windows.md:

import IframeVideo from '@site/src/components/IframeVideo/index';

Hacker house - initial Nillion task - Windows

👋 Hey, welcome to Nillion. We are really pleased to have you start your journey with us.

This page describes the initial Nillion task for Hacker House Goa for Windows users. If you are on MacOS or Linux, head over to this page.

The initial task should take 30-45 minutes and will help you get started with Nillion. Join our social channels, and start working with our SDK. If you have any questions, feel free to ask them on the discord channel.

The 50 best tasks that are submitted will be awarded a $20 prize 🎉 Good luck!

Follow the steps below to complete the task:

1. Join our community:
    - Join our Discord server
      - Make sure you join the developers-sdk channel.
      - Once you fill out the form at the end of this task, we will add you to a dedicated hacker-house-goa Discord channel too.
    - Follow us on Twitter

2. Star & fork the Python quickstart repo & the JavaScript Quickstart repo.

    📥 Add the topic nillion-nada to this repo so we can find it easily

3. Head to this Google Colab and complete the initial Nillion task
   - All instructions are in the Google Colab, make sure you read them carefully, and follow the steps in order.
   - There is a $20 prize for the best 50 new nada programs that are written in step 9.

4. Fill out this form to complete the initial task.

# hacker-house-goa.md:

import IframeVideo from '@site/src/components/IframeVideo/index';

Hacker house - initial Nillion task - MacOS & Linux

👋 Hey, welcome to Nillion. We are really pleased to have you start your journey with us.

This page describes the initial Nillion task for Hacker House Goa for MacOS and Linux users. If you are on Windows, head over to this page.

The initial task should take 30-45 minutes and will help you get started with Nillion; join our social channels, and start working with our SDK. If you have any questions, feel free to ask them on the discord channel.

The 50 best tasks that are submitted will be awarded a $20 prize 🎉 Good luck!

Follow the steps below to complete the task:

1. Join our community:

   - Join our Discord server
     - Make sure you join the developers-sdk channel
     - Once you fill out the form at the end of this task, we will add you to a dedicated hacker-house-goa Discord channel too.
   - Follow us on Twitter

2. Star & fork the Python quickstart repo & the JavaScript Quickstart repo.

    🥷 Add the topic nillion-nada to this repo so we can find it easily

3. Follow at least one of the quickstarts (Python or JavaScript) in the Developer Quickstart

    🥷 Make sure you enable telemetry as you install the Nillion SDK. Start your telemetry identifier with hacker-house-goa so we can identify you

4. Once you have completed the quickstart, add at least one new nada program to your repo (bonus points for creativity, but make sure it compiles and runs)

   - Python quickstart: add your nada program in your repo's programs folder.
   - JavaScript quickstart: add your program in your repo's public/programs folder
   - The Nillion team will review all your new nada programs and award $20 prizes to the best 50.

5. Fill out this form to complete the initial Nillion task for Hacker House Goa.

# high-value-data.md:

---
description: If you care about it, then it's probably high value data (HVD).
---

High Value Data (HVD)

High value data refers to data that is extremely valuable to an organization or individual due to its potential to significantly impact everyone’s lives – from AI to trading information (leverage and limit orders) to identity to healthcare data to access control to decentralized social to passwords to biometrics – high value data is woven into the infrastructure of our society.

Nillion is decentralizing HVD using cryptography, not just as a defensive strategy against threats, but as a proactive step toward new use cases and a fairer data ecosystem. To learn more, read our Chief Scientist’s blog post:

Introducing A High Value Data Vision For The Future.

# installation.md:

```
import SdkInstallation from './\sdk-installation.mdx';
import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
```

Install Nillion SDK and Tools

```
<Tabs>
    <TabItem value="mac-linux-guide" label="Mac and Linux" default>
        Mac and Linux Guide

        <SdkInstallation/>

    </TabItem>

    <TabItem value="windows-guide" label="Windows">
        Windows Guide
```

Today Nillion SDK binaries are available for Mac and Linux. In order to install Nillion on a Windows machine, you'll need to first complete a 5 minute WSL developer environment setup. Follow the steps below to install WSL, set up your WSL developer environment, and install and use Nillion Linux binaries from a Windows machine.

WSL developer environment setup

The Windows Subsystem for Linux (WSL) lets developers install a Linux distribution and use Linux applications, utilities, and Bash command-line tools including the Nillion SDK and tools on a Windows machine.

1. Run the install command

From your terminal, install WSL

```
wsl --install
```

This downloads and installs the Ubuntu Linux distribution.

2. Restart your computer

A reboot is required after installing Ubuntu.

3. After restarting, open Ubuntu and create an account

To open an Ubuntu terminal, search for the app from Start. Open Ubuntu and follow the prompts to set up your Linux username and password.

4. Update and upgrade packages

```
sudo apt update && sudo apt upgrade
```

5. Install the Visual Studio Code Remote Development Extension

Visual Studio Code Remote Development allows you to use a container, remote machine, or the Windows Subsystem for Linux (WSL) as a full-featured

```
development environment. Install the Visual Studio Code Remote Development
Extension Pack.

        6. Open a new Ubuntu terminal and Install Nillion

        Now that your WSL development environment is set up, you can install the
Nillion SDK and Tools.

        :::info
        Make sure to install Nillion within a new Ubuntu terminal. Either open a
Ubuntu terminal, or from your Windows PowerShell terminal, first run Ubuntu:


        ubuntu


        :::

        <div className="divider"/>
        Installation


        <SdkInstallation />


        🎉 Great work! You've set up a WSL environment and installed Nillion
within that environment. Before running a nillion command like nillion-devnet,
nada, or pynadac make sure you are in a WSL environment by first running ubuntu:


        ubuntu


    </TabItem>

</Tabs>


# js-client-examples.md:

import DocCardList from '@theme/DocCardList';

JavaScript Client Examples

CRA-Nillion Starter Repo

The CRA-Nillion Starter Repo used in the JavaScript Developer Quickstart
contains storage, retrieval, compute, and permissions examples.

<DocCardList />


# js-client-reference.md:

JavaScript Client Reference

@nillion/client-web

> Javascript Nillion Client

addcomputepermissions(compute)

Add compute permissions to the :py:class:Permissions instance for the
given list of user IDs
```

Arguments

compute : dict of list of str
Dict keyed by the userid of the targets where the value is a list of str
specifying which program IDs to permit compute for

Example

    TODO

Parameters

Name	Types	Description
compute	any	

adddeletepermissions(permissions)

Add delete permissions to the Permissions instance for the
given list of user IDs

Example

    TODO

Parameters

Name	Types	Description
permissions	any	

addinputparty(name, id)

Bind an input party with a name

Parameters

Name	Types	Description
name	string	
id	string	

addoutputparty(name, id)

Bind an output party with a name

Parameters

Name	Types	Description
name	string	
id	string	

addretrievepermissions(permissions)

Add retrieve permissions to the Permissions instance for the
given list of user IDs

Example

    TODO

Parameters

Name	Types	Description
permissions	any	

addupdatepermissions(permissions)

Add update permissions to the Permissions instance for the
given list of user IDs

Example

```
TODO
```

Parameters

Name	Types	Description
permissions	any	

clusterinformation(clusterid)

Returns information about a Nillion cluster

Arguments

- clusterid - UUID of the targeted preprocessing cluster

Returns

The cluster descriptor for the given cluster

Parameters

Name	Types	Description
clusterid	string	

Returns

Promise.&lt;ClusterDescriptor&gt;

compute(clusterid, bindings, storeids, secrets, publicvariables)

Compute in the Nillion Network

This method invokes the compute operation in Nillion.
It returns a compute ID that can be used by computeresult to fetch
the results of this computation.

Parameters

Name	Types	Description
clusterid	string	UUID of the targeted preprocessing cluster
bindings	ProgramBindings	The program bindings for the computation
storeids	Array.&lt;string&gt;	The store IDs of the secrets to use for the computation
secrets	Secrets	Additional secrets to use for the computation

| publicvariables | PublicVariables | Public variables that are used in the computation. |

Returns

Object
A computation UUID.

computeresult(resultid)

Fetch the result of the compute in the Nillion Network

Arguments

- resultid - computation UUID

Returns

A computed JsValue

Parameters

Name	Types	Description
resultid	string	

Returns

Promise.&lt;any&gt;

defaultforuser(userid, program)

Returns the default permission set for the given user ID

Arguments

userid : str
Desired target peer ID
program : str, optional
Specify a Program ID to apply permission to.

Returns

Permissions

Example

```
    permissions = nillion.Permissions.defaultforuser(client.userid)
```

Parameters

Name	Types	Description
userid	string	
program	string	

Returns

Permissions

deletesecrets(clusterid, storeid)

Delete secrets collection from the network

Arguments

- clusterid - The cluster identifier
- storeid - The store operation identifier for the secret collection

Returns

The unique identifier of the delete operation

Parameters

Name	Types	Description
clusterid	string	
storeid	string	

Returns

Promise.&lt;void&gt;

enabletracking(walletaddr)

Enables tracking for the user

Arguments

Returns

Enables tracking of client actions (store, retrieve, compute ...)

Parameters

Name	Types	Description
walletaddr	string, undefined	

frombase58(contents)

Decodes a private key from a string encoded in Base58.

Arguments

- contents - A base58 string

Returns

An instance of NodeKey matching the string provided

Parameters

Name	Types	Description
contents	string	

Returns

NodeKey

frombase58(contents)

Loads a [UserKey] from a Base58 String

Arguments

- contents - The private key encoded in Base58 format

Parameters

Name	Types	Description
contents	string	

Returns

UserKey

fromseed(seed)

Generates a private key using a seed.

Returns

A NodeKey

Parameters

Name	Types	Description
seed	string	

Returns

NodeKey

fromseed(seed)

Generate a new public/private key.
Uses a seed to generate the keys.

Arguments

- seed - The seed as a [&str]

Returns

A [Result] whose Ok value is the [UserKey] generated using the seed provided

Parameters

Name	Types	Description
seed	string	

Returns

UserKey

generate()

Generate a new random public/private key.
Uses a cryptographically secure pseudo-random number generator.

Returns

UserKey

insert(name, input)

Add encoded public variable to the PublicVariables collection.

Parameters

Name	Types	Description
name	string	
input	PublicVariable	

insert(name, input)

Add encoded secret to Secrets collection.

Parameters

Name	Types	Description
name	string	
input	Secret	

iscomputeallowed(userid, program)

Returns true if user has compute permissions for every single program

Returns

bool

Example

```
TODO
```

Parameters

Name	Types	Description
userid	string	
program	string	

Returns

boolean

isdeleteallowed(userid)

Returns true if user has delete permissions

Returns

bool

Example

```
TODO
```

Parameters

Name	Types	Description
userid	string	

Returns

boolean

isretrieveallowed(userid)

Returns true if user has retrieve permissions

Returns

bool

Example

```
    TODO
```

Parameters

Name	Types	Description
userid	string	

Returns

boolean

isretrievepermissionsallowed(userid)

Checks if user is allowed to retrieve the permissions

Returns

bool

Example

.. code-block:: py3

```
    TODO
```

Parameters

Name	Types	Description
userid	string	

Returns

boolean

isupdateallowed(userid)

Returns true if user has update permissions

Returns

bool

Example

```
    TODO
```

Parameters

Name	Types	Description
userid	string	

Returns

boolean

isupdatepermissionsallowed(userid)

Checks if user is allowed to update the permissions

Returns

bool

Example

```
    TODO
```

Parameters

Name	Types	Description
userid	string	

Returns

boolean

mainJS()

null

Returns

string

newblob(value)

Create a new secret blob with the provided value.

Parameters

Name	Types	Description
value	Uint8Array	

Returns

Secret

newinteger(value)

Create a new public integer with the provided value.

The value must be a valid string representation of an integer.

Parameters

Name	Types	Description
value	string	

Returns

PublicVariable

newinteger(value)

Create a new secret integer with the provided value.

The value must be a valid string representation of an integer.

Parameters

Name	Types	Description
value	string	

Returns

Secret

newunsignedinteger(value)

Create a new public unsigned integer with the provided value.

The value must be a valid string representation of an unsigned integer.

Parameters

Name	Types	Description
value	string	

Returns

PublicVariable

newunsignedinteger(value)

Create a new secret unsigned integer with the provided value.

The value must be a valid string representation of an unsigned integer.

Parameters

Name	Types	Description
value	string	

Returns

Secret

partyid

Get partyid property

Arguments

Returns

A party id

Returns

string

publickey()

Returns the public key corresponding to this key.

Returns

The public key as an UTF-8 encoded string.

Returns

string

retrievepermissions(clusterid, storeid, secretid)

Retrieve permissions from nillion

Arguments

- clusterid - The cluster identifier
- storeid - The store operation identifier for the secret
- secretid - The identifier of the secret

Returns

The permissions as [NilPermissions]

Parameters

Name	Types	Description
clusterid	string	
storeid	string	
secretid	string	

Returns

Promise.&lt;Permissions&gt;

retrievesecret(clusterid, storeid, secretid)

Retrieve a secret on the network with javascript

Arguments

- clusterid - UUID of the targeted preprocessing cluster
- storeid - The secret's store ID (returned when calling storesecrets)
- secretid - The secret's ID

Returns

The secret ID as a UUID as well as the secret itself as Secret

Parameters

Name	Types	Description
clusterid	string	
storeid	string	
secretid	string	

Returns

Promise.&lt;Secret&gt;

storeprogram(clusterid, programname, program)

Store a program in the Nillion Network

Arguments

- clusterid - UUID of the targeted preprocessing cluster
- programname - The name of the program
- program - The compiled program

Returns

The action ID associated with the action of storing a program

Parameters

Name	Types	Description
clusterid	string	
programname	string	
program	Uint8Array	

Returns

Promise.&lt;string&gt;

storesecrets(clusterid, secrets, bindings, permissions)

Store a secret on the network from javascript; clear envelope

Arguments

- clusterid - UUID of the targeted preprocessing cluster
- secrets - The NilSecrets collection of secrets to store
- bindings - Optional bindings between network parties and the parties are defined by the program

Returns

A store ID that can be used to retrieve the secret.

Parameters

Name	Types	Description
clusterid	string	
secrets	Secrets	
bindings	ProgramBindings, undefined	
permissions	Permissions, undefined	

Returns

Promise.&lt;string&gt;

tobase58()

Returns the key in Base58 encoded form.

Returns

string

tobytearray()

Convert this secret into a byte array.

This is only valid for blob secrets.

Returns

Uint8Array

tointeger()

Convert this public variable into a string representation of the underlying numeric value.

This only works for numeric public variables, such as integers and unsigned integers.

Returns

string

tointeger()

Convert this secret into a string representation of the underlying numeric value.

This only works for numeric secrets, such as integers and unsigned integers.

Returns

string

updatepermissions(clusterid, storeid, secretid, permissions)

Update permissions from nillion

Arguments

- clusterid - The cluster identifier
- storeid - The store operation identifier for the secret
- secretid - The identifier of the secret
- permissions - The permissions that will replace the existing permissions for the secret

Returns

The Update Permissions action ID

Parameters

Name	Types	Description
clusterid	string	
storeid	string	
secretid	string	
permissions	Permissions	

Returns

Promise.&lt;string&gt;

updatesecrets(clusterid, storeid, secrets, bindings)

Update a secret stored in nillion

Arguments

- clusterid - The cluster identifier
- storeid - The store operation identifier for the secret
- secrets - The secrets to be updated

Returns

The unique identifier of the update operation

Parameters

Name	Types	Description
clusterid	string	
storeid	string	
secrets	Secrets	
bindings	ProgramBindings, undefined	

Returns

Promise.&lt;string&gt;

userid

Get the user ID of the Client instance.

Arguments

Returns

An user identifier

Returns

string

workerentrypoint(sharedstateptr)

Entry point invoked by the web worker. The passed pointer will be
unconditionally interpreted
as an Box::<(WorkerExecutor, Worker)>.

Parameters

Name	Types	Description
sharedstateptr	number	

# js-client.md:

```
import DocCardList from '@theme/DocCardList';
import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import JsHeaders from './\js-headers-proxy.mdx';
```

JavaScript Client

The @nillion/client-web npm package is a JavaScript client for building on top of the Nillion Network. It can be used to manage Nada programs, store and retrieve secrets, and run computations.

Installation

```
Install the JavaScript Client package in your browser application.

<Tabs>

  <TabItem value="yarn" label="yarn" default>
bash
yarn add @nillion/client-web

  </TabItem>

  <TabItem value="npm" label="npm">
bash
npm i @nillion/client-web

  </TabItem>

  <TabItem value="pnpm" label="pnpm">
bash
pnpm add @nillion/client-web

  </TabItem>
</Tabs>

Usage

Import JavaScript Client

js
import  as nillion from '@nillion/client-web';


Set Headers and set up proxy for nilchain

<JsHeaders/>

Initialize NillionClient with JavaScript Client

<Tabs>

<TabItem value="app" label="App.js" default>
js
import  as nillion from '@nillion/client-web';
import React, { useState, useEffect } from 'react';

const App: React.FC = () => {
const [nillionClient, setNillionClient] = useState(null);
const userKeySeed = 'my-userkey-seed';
const nodeKeySeed = my-nodekey-seed-${Math.floor(Math.random()  10) + 1};

const initializeNewClient = async () => {
if (userKey) {
await nillion.default();
const uk = nillion.UserKey.frombase58(userKey);
const nodeKey = nillion.NodeKey.fromseed(nodeKeySeed);
const userKey = nillion.UserKey.fromseed(nodeKeySeed);
const newClient = new nillion.NillionClient(userkey, nodeKey,
process.env.REACTAPPNILLIONBOOTNODEWEBSOCKET);
setNillionClient(newClient);
}
};

useEffect(() => {
initializeNewClient();
}, []);
return (

<div className="App">
<h1>YOUR APP HERE</h1>
User ID: {nillionClient ? nillionClient.userid : 'Not set - Nillion Client has
not been initialized' }
</div>
);
};

export default App;


</TabItem>

<TabItem value="env" label=".env" default>
```

Populate the .env with your Nillion Network config - here's an example of a .env file populated with a local nillion-devnet configuration

```
txt
replace with values from nillion-devnet

REACTAPPNILLIONCLUSTERID=
REACTAPPNILLIONBOOTNODEWEBSOCKET=
REACTAPPNILLIONNILCHAINJSONRPC=
REACTAPPNILLIONNILCHAINPRIVATEKEY=
REACTAPPAPIBASEPATH=/nilchain-proxy


</TabItem>
</Tabs>
```

Resources

<DocCardList/>

# js-quickstart.md:

```
import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import IframeVideo from '@site/src/components/IframeVideo/index';
import SdkInstallation from './\sdk-installation.mdx';
import QuickstartIntro from './\quickstart-intro.mdx';
import VenvSetup from './\nada-venv-setup.mdx';
import UnderstandingProgram from './\understanding-first-nada-program.mdx';
import CompileRunTest from './\quickstart-compile-run-test.mdx';
import JsHeaders from './\js-headers-proxy.mdx';
```

JavaScript Developer Quickstart

Welcome to the JavaScript Quickstart. By the end of this guide, you will have:

1. Installed the Nillion SDK and set up your dev environment
2. Written, compiled, and tested your first nada program using the nada tool
3. Plugged your nada program into your first 'blind app' with starter code in cra-nillion
4. Connected your blind app to your local nillion-devnet to run it locally

Once you have finished, explore other demo pages in the cra-nillion repo to continue your Nillion developer journey!

Install the Nillion SDK tools

<SdkInstallation/>

Create a new folder for your quickstart

Create quickstart, a folder to hold your quickstart project - by the end of the quickstart, the folder will contain 2 subfolders - one for your nada project and one for the cloned cra-nillion starter repo

```
mkdir quickstart
```

Create a new Nada project

```
cd quickstart
nada init nadaquickstartprograms
```

This will create a directory called nadaquickstartprograms, which is your Nada project. You can read more about the file structure of this new Nada project here

Set up virtual environment

:::info

We're still in the JavaScript Quickstart, but in order to run the Nada program we're about to write, you'll need python3 version 3.11 or higher with a working pip installed

- Confirm that you have python3 (version >=3.11) and pip installed
```
     python3 --version
  python3 -m pip --version
```

  :::

Change directories into your new Nada project

```
cd nadaquickstartprograms
```

<VenvSetup/>

Your first program

The code for the finished program is below - it is a simple program that has one party and adds two secret integer inputs together.

```
python
from nadadsl import

def nadamain():

    party1 = Party(name="Party1")

    myint1 = SecretInteger(Input(name="myint1", party=party1))

    myint2 = SecretInteger(Input(name="myint2", party=party1))
    newint = myint1 + myint2

    return [Output(newint, "myoutput", party1)]
```

Now we will write it from scratch, explaining how it works as we go. Once we have written the program, we will use the nada tool to run and test it.

1. From the nadaquickstartprograms Nada project, cd into the src folder and create a program file:
```
   bash
   cd src
   touch secretaddition.py
```

2. Write or copy the program above into this file

Understanding the program you have just written

<UnderstandingProgram/>

Compile, run and test your program

Make sure you are in the quickstart/nadaquickstartprograms directory.

<CompileRunTest/>

Well done! You've just written and tested your first Nada program! Now we'll hook this up to a blind app, which will be able to compute with the secretaddition Nada program on secret inputs.

Clone the CRA-Nillion JavaScript starter repo

The cra-nillion Starter Repo repo is a Create React App which has everything you need to start building your blind app. Clone the repo:

Make sure you are in the root of the quickstart directory.

```
bash
git clone https://github.com/NillionNetwork/cra-nillion.git
```

Install repo dependencies and run the starter

:::info

Before you use cra-nillion, check that you have Node (>= v18.17) installed by running
```
   node -v
```

:::

```
cd cra-nillion
npm i
npm start
```

Open http://localhost:8080/ to see your cra-nillion starter app running locally at port 8080

!CRA nillion no cluster

For this Quickstart, we'll focus on the Nillion Operations page and the Nillion Blind Computation Demo page.

Connect the blind app to nillion-devnet

In the screenshot of cra-nillion, you'll notice that cluster id and other configuration variables needed to connect to the Nillion Network are not set, so it's not possible to connect NillionClient.

Spin up a local Nillion devnet

Open a second terminal and run the devnet using any seed (the example uses "my-seed") so the cluster id, websockets, and other environment variables stay constant even when you restart nillion-devnet.

```
shell
nillion-devnet --seed my-seed
```

You will see an output like this:

```
nillion-devnet --seed my-seed
ℹ️ cluster id is 222257f5-f3ce-4b80-bdbc-0a51f6050996
ℹ️ using 256 bit prime
ℹ️ storing state in /var/folders/1/2yw8krkx5q5dn2jbhx69s4r0000gn/T/.tmpU00Jbm
(62.14Gbs available)
🏃 starting nilchain node in:
/var/folders/1/2yw8krkx5q5dn2jbhx69s4r0000gn/T/.tmpU00Jbm/nillion-chain
⛓  nilchain JSON RPC available at http://127.0.0.1:48102
⛓  nilchain gRPC available at localhost:26649
🏃 starting node 12D3KooWMGxv3uv4QrGFF7bbzxmTJThbtiZkHXAgo3nVrMutz6QN
⏳ waiting until bootnode is up...
🏃 starting node 12D3KooWKkbCcG2ujvJhHe5AiXznS9iFmzzy1jRgUTJEhk4vjF7q
🏃 starting node 12D3KooWMgLTrRAtP9HcUYTtsZNf27z5uKt3xJKXsSS2ohhPGnAm
👛 funding nilchain keys
📝 nillion CLI configuration written to /Users/steph/Library/Application
Support/nillion.nillion/config.yaml
🌄 environment file written to /Users/steph/Library/Application
Support/nillion.nillion/nillion-devnet.env
```

Copy the path printed after "🌄 environment file written to" and open the file

```
vim "/Users/steph/Library/Application Support/nillion.nillion/nillion-devnet.env"
```

This file has the nillion-devnet generated values for cluster id, websocket, json rpc, and private key. You'll need to put these in your local .env in one of the next steps so that your cra-nillion demo app connects to the nillion-devnet.

Keep the nillion-devnet running in this terminal.

Create .env file

Make sure you are in the quickstart/cra-nillion directory.

Copy the up the .env.example file to a new .env and set up these variables to match the nillion environment file.

```
shell
cp .env.example .env
```

Update your newly created .env with environment variables outout in your terminal by nillion-devnet

```
REACTAPPNILLIONCLUSTERID=
REACTAPPNILLIONBOOTNODEWEBSOCKET=
REACTAPPNILLIONNILCHAINJSONRPC=
REACTAPPNILLIONNILCHAINPRIVATEKEY=
REACTAPPAPIBASEPATH=/nilchain-proxy
```

Restart the cra-nillion app process

```
npm start
```

Now the Cluster ID field should be populated with the nillion-devnet cluster id value you set in REACTAPPNILLIONCLUSTERID.

!CRA nillion with cluster

Try out the Operations Page

1. Generate User Key - generates a new user key / user id pair
2. Connect with User Key - sets the user key and connects to NillionClient via the Nillion JavaScript Client
3. Hide Nillion User Key and Node Key Connection Info - toggle button to show/hide user and node key options
4. Perform Nillion Operations

To perform an operation (store secret, retrieve secret, update secret, store program, compute), you follow the same pattern:

1. Get quote for the operation
2. Pay quote for the operation and get a payment receipt. On your local nillion-devnet, payments for operations are sent to the local nilchain at REACTAPPNILLIONNILCHAINJSONRPC are funded by REACTAPPNILLIONNILCHAINPRIVATEKEY
3. Perform the operation with the payment receipt as a parameter

   !CRA nillion operations

Hook up your secretaddition.py nada program to your first blind app

Now that you understand how Nillion operations work, let's update the Nillion Blind Computation Demo page to use the Nada program you created.

Navigate to the Blind Computation Demo page: http://localhost:8080/compute

The code for Blind Computation Demo page lives in ComputePage.tsx

```
const outputName = 'myoutput';
const partyName = 'Party1';
```

Notice that the ComputePage sets outputName which matches the the output name set in secretaddition.py. The ComputePage sets partyName which matches the the party name set in secretaddition.py. There are 2 StoreSecretForm components on ComputePage, with secretName set to myint1 and myint2 which matches the the secret names set in secretaddition.py.

<Tabs>
<TabItem value="helpers" label="secretaddition.py" default>

```python
from nadadsl import

def nadamain():

    party1 = Party(name="Party1")

    myint1 = SecretInteger(Input(name="myint1", party=party1))

    myint2 = SecretInteger(Input(name="myint2", party=party1))

    newint = myint1 + myint2

    return [Output(newint, "myoutput", party1)]
```

</TabItem>

<TabItem value="compute" label="ComputePage.tsx" >
ts reference showGithubLink
https://github.com/NillionNetwork/cra-nillion/blob/main/src/ComputePage.tsx

</TabItem>

</Tabs>

Update programName

In order to run blind computation on your secretaddition.py Nada program, you'll need to make a few updates:

1. Open a new terminal and navigate to the root quickstart folder. List the contents of the folder

```
ls
```

You should see cra-nillion and nadaquickstartprograms folders.

2. Copy your secretaddition.py and secretaddition.nada.bin files nadaquickstartprograms into cra-nillion

```
cp nadaquickstartprograms/src/secretaddition.py cra-nillion/public/programs
cp nadaquickstartprograms/target/secretaddition.nada.bin
cra-nillion/public/programs
```

Now your cra-nillion app can use the nada program and the nada program binaries in store program operations.

3. Update programName to secretaddition so the cra-nillion repo reads your Nada program.

ts reference showGithubLink
https://github.com/NillionNetwork/cra-nillion/blob/main/src/ComputePage.tsx#L13

Run the Blind Computation Demo

Go back to the Blind App on http://localhost:8080/compute and run through the steps on the page to test out the full blind computation flow.

Keep exploring

You've successfully build your first blind app by writing a Nada program, storing the program on the network, storing secrets on the network, and running compute against secrets. Keep exploring by

- reading about Nillion concepts and the Nada Language
- learning how to interact with and manage programs, secrets, and permissions on the Nillion Network with Nillion Client
- challenging yourself to create a page that solves the Millionaires Problem

:::tip

Open the Nillion JavaScript Client Reference doc in another tab to search for available Nillion Client classes while working with cra-nillion.

:::

# limitations.md:

Limitations

Our SDK release provides developers with the chance to start building with Nillion. The current release allows developers to build with our Python, JavaScript and CLI clients, Nada language, and our command line tools. As we build the Nillion SDK and tools in public, we want to work with the community to make them even better.

Expectations

- Evolving UX, developer tools, and documentation
- Frequent, sometimes breaking changes, but we will do our best to communicate them ahead of time in Announcements
- Bugs - please report any you find in Bugs
- Missing features - suggest a feature request in Ideas
- Documentation gaps - if you notice something is missing, please let us know by creating a Github Issue

Here are some of the limitations or constraints you should be aware of when being an early builder with Nillion’s SDK.

Platforms

The Nillion SDK tool binaries can be installed with nilup for the following platforms:

- Apple (M1/M2)
- Apple (Intel chip)
- Linux (ARM chip)
- Linux (Intel/AMD chip)

If you are on a Windows machine, follow our Windows guide to set up a WSL developer environment and install Linux binaries.

Nillion Clients

- We have released 3 clients to the community: a Python, JavaScript, and CLI Client. In general the clients have feature parity, however please refer to the Nillion Client docs to see the exact functionality provided by each.
- You can only compile programs from the CLI pynadac or nada tools — compilation isn’t available in the Python or JavaScript clients.
- If there is a particular feature you need and believe is missing in one of the

clients, please report it in Bugs.

JavaScript Client

- Currently the JS client is only tested on Chromium browsers (Chrome, Brave & Edge) and production deployments will likely require activating COOP and COEP headers.
- The JS client is a browser client and does not yet work in NodeJs.

CLI Client

Running the nillion-devnet command (SDK tool) will spin up a local devnet on your machine.

- The devnet that is spun up will be limited by the hardware it is running on. Keep an eye on the CPU usage when running large computations.
- Pre-processing elements are generated from scratch each time a local devnet is spun up. This means you may need to wait a little time (10-15 seconds) before the network is able to store and compute. For reference, currently 8192 alphas & 8192 lambdas are generated when you spin up a new local devnet.
- Anything you store (programs, secrets etc) in one instance of nillion-devnet will not be shared with another devnet you spin up, meaning you will have to restore any stored programs or secrets in the new devnet.
- The devnet does not currently support transport targets other than localhost.

One node key per client

- If you are instantiating multiple clients (users) at once (e.g. Party1, Party2, Party3 for a multi party compute), ensure that each instantiated client uses a different node key. If a node key is reused, across clients, you will see a timeout error.

Secrets

- Secrets are not yet user scoped.
- Continuously storing secrets in a loop can lead to errors due to the necessity of blinding factors for secret storage. Blinding factors are generated by the network on demand when a threshold is met. If secrets are stored continuously, the network may struggle to generate blinding factors in time.

Nada language, programs, and inputs

Check the Nada language docs for the current data types and operations available.

No random value generation in Nada

- We expect random value generation in Nada in the future.

No function composition / Simplified scope

Function composition is not yet possible due to Nada's simplified variable and scope management. Consider the following code fragment:

```
python
def inc(a: SecretInteger, myint: SecretInteger) -> SecretInteger:
    return a + myint

def inc2(a: SecretInteger) -> SecretInteger:
    return inc(a, SecretInteger(2))
```

The Nada compiler will throw an error as inc2 does not have access to inc.

Zeros

- Ensure that your programs do not attempt to divide by 0, this is not supported in our language currently.

Run a testnet node

Developers can run a local Nillion Network node with the nillion-devnet tool.

Today, running a Nillion Network Testnet node is permissioned. We plan to transition to a permissionless model in the future, allowing the public to participate as node operators. When this changes, you’ll hear about it on our Twitter and Discord. Follow us to stay updated.

# multi-party-computation.md:

```
---
description: >-
  Multi-Party Computation, or MPC, is a privacy enhancing technology (PET) with
  the potential to transform how we secure and compute on high value data in
  collaborative environments.
---
```

Multi-Party Computation (MPC)

Traditional data handling follows a decrypt-compute-reencrypt cycle: data has to be decrypted in order to run a computation, and then re-encrypted for storage or transmission. Decrypted data is vulnerable to unauthorized access and potential security breaches, exposing sensitive information during the computation phase.

MPC enables computation over inputs from multiple parties while keeping the actual data hidden. The data remains confidential throughout the entire process — not only when it's stored or in transit, but also during the actual computation. With MPC, you can collaborate to compute on hidden secrets without ever revealing the actual secrets.

Applied MPC

Classic Scenario: The Millionaire Problem

The "Millionaires Problem" is a classic MPC scenario, first introduced by Andrew Yao in 1982. In the "Millionaires Problem" problem, two millionaires want to find out who is richer without disclosing their actual net worth. Using MPC, the millionaires can jointly compute who has more money without revealing their individual net worths to each other or anyone else. This is achieved through a series of cryptographic operations that allow each party to input their net worth into a shared computation. The computation is structured in a way that it only outputs the comparison result (i.e., which millionaire is richer) without leaking any specifics about their respective net worths. This problem showcases the power of MPC - it can preserve privacy while enabling collaborative computation.

Real World Application: The 2008 Danish Sugar Beet Auction

In the early 2000s, Denmark's sugar industry faced a critical challenge: how to conduct auctions for sugar beet contracts securely while maintaining confidentiality. Traditional auction methods required bidders to openly reveal their bids, leading to concerns about fairness and the potential for bid manipulation. To address these issues, the industry sought a solution that would enable secure bidding without compromising the privacy of individual bids.

Enter secure MPC. By leveraging MPC, farmers could submit encrypted bids to processors, ensuring that bid information remained confidential throughout the auction. This innovative approach not only safeguarded the integrity of the bidding process but also promoted fairness and transparency, setting a new standard for auction security and confidentiality in the sugar industry. The adoption of MPC led to a notable increase in beet farmer satisfaction. 80% of farmers surveyed emphasized the significance of maintaining bid confidentiality during the bidding. Responses indicated high levels of confidence in the fairness and confidentiality of the auction process, further solidifying the benefits of MPC in real-world applications. Read more about the Danish Sugar Beet Auction implementation in Secure Multiparty Computation Goes Live.

Additional MPC Resources

- Learn about MPC, a set of resources curated by the MPC Alliance
- awesome-mpc Github list

# nada-ai-introduction.md:

Nada AI

nada-ai is a cutting-edge Machine Learning framework inspired by SKLearn and PyTorch, designed to seamlessly integrate model-building capabilities into Nada DSL. With Nada AI, effortlessly import AI models into the Nillion ecosystem for inference, leveraging the robust foundation of Nada Numpy.

Nada AI boasts a strongly-typed interface, utilizing root families (Integer, UnsignedInteger, Rational, and SecretBoolean). This ensures strict type enforcement at the library level, guaranteeing compliance with Nada DSL rules.

LLMs and Nada AI

Choosing blind computing involves certain trade-offs compared to traditional computing. While you gain enhanced privacy, there are added computational overheads and capacity constraints.

Currently, Nada-AI does not support Large Language Models (LLMs). However, we are actively working on integrating this capability into our platform. In the meantime, follow our 3-phase workflow with a supported model to build a blind AI project with Nada:

Discover the Power of Privacy-Preserving AI

In this tutorial series, you'll dive into creating your own privacy-preserving AI models for inference. Developing a model with Nillion involves three distinct phases:

1. Train Your Plaintext Model: Use your dataset to train a model with your preferred AI tools. Nada AI currently supports Scikit-Learn, various layers in PyTorch, and the Prophet time series model.
2. Write Your Nada Program: Develop the Nada code to be executed in MPC, utilizing the bridges provided by Nada AI. You'll find it strikingly similar to the original Python code.
3. Store and Run Your Program: Compile the Nada code, store it on Nillion, and execute it on the network to obtain live, privacy-preserving predictions from your model.

Explore our GitHub Repository Examples for hands-on learning and insights.

Supported Models

- Multilayer Perceptron: With the following layers available: Linear, Conv2d, AvgPooling2d, DotProductSimilarity, ReLU, Flatten.
- Linear Regression Model: Linear model.
- Logistic Regression Model: Linear model implementation with cleartext sigmoid and potential multiclass classification.

- Prophet: Time series forecasting model.

Important Considerations and Limitations

Nada AI is currently in active development. Please note:

- AI training is not supported at this time. Both MPC and AI training are computationally intensive tasks requiring specific setups for efficient use. The current roadmap includes support for Federated Learning and other training procedures for privacy-preserving AI training.
- The described workloads have been tested on the Nillion devnet tool. Due to high demand, running large models on the Nillion Testnet or Mainnet may not always be possible. For specific use cases or if you wish to run models on the Testnet or Mainnet, please contact us here.

Embark on your journey with Nada AI and revolutionize your approach to privacy-preserving machine learning!

# nada-ai-linear-model.md:

Nada AI Linear Model Tutorial

In this tutorial, we’ll explore the capabilities of Nada AI to create a basic Linear Regression model for inference. First, we'll train our model, and then import it to Nada and the Nillion Network.

Before starting, make sure you have the Nillion SDK installed.

Installation

First things first, let’s get Nada Numpy installed on your system. It’s as easy as running:

```
bash
pip install --upgrade nada-ai
```

Training our Linear Regression Model

If you've previously installed nada-ai, you should already have the dependencies required for this demo. For simplicity, we'll be training on random inputs and outputs. You can find the complete training script here:

```
python
import numpy as np
from sklearn.linearmodel import LinearRegression

Create a random input dataset of 1000 samples with 10 features per sample
X = np.random.randn(1,000, 10)

We assume trained coefficients are the data from a specific linear model
expectedweights = np.ones((10,))
expectedbias = 4.2  Value taken randomly

y = X @ expectedweights + expectedbias

model = LinearRegression()
The learned parameters will likely be close to the coefficients & bias we used
to generate the data
fitmodel = model.fit(X, y)
```

Now, let's break down the different parts of the program. First, we import the LinearRegression model from sklearn to be used in our training. Furthermore, we'll use Numpy to produce a random input dataset.

python
```
import numpy as np
from sklearn.linearmodel import LinearRegression
```

We create a random example in a particular way. First, we produce X, which consists of 1000 random samples with 10 features each. Then, we generate our expectedweights and expectedbias and use those to generate the output y. In this way, we assume that training on X and y should yield the expectedweights and expectedbias as a result.

python
```
Create a random input dataset of 1000 samples with 10 features per sample
X = np.random.randn(1,000, 10)

We assume trained coefficients are the data from a specific linear model
expectedweights = np.ones((10,))
expectedbias = 4.2  Value taken randomly

y = X @ expectedweights + expectedbias
```

Then we can proceed to train our Linear Regression model. The line below first initializes the LinearRegression model and then calls the fit function to proceed with the training.

python
```
model = LinearRegression()
fitmodel = model.fit(X, y)
```

Based on the dataset creation, the learned weights and bias should resemble the expectedweights and expectedbias. These can be extracted to be imported in the next sections. Whether you save them to a .json file, or copy-paste them is up to the user.

python
```
weights = fitmodel.coef
bias = fitmodel.intercept
```

Now that we've trained our model, we can proceed to migrate it to Nillion for private execution.

Writing our Nada-AI program

To get started, initialize your project structure with the following command:
bash
```
nada init nada-linear-regression
```

This sets up your environment. We'll create a program that performs linear regression inference. Here’s the complete code to place in src/main.py:

python
```
from nadadsl import
import nadanumpy as na
from nadaai.linearmodel import LinearRegression

def nadamain():
    Step 1: Define the parties involved in our computation
    user = Party(name="User")
    provider = Party(name="Provider")

    Step 2: Instantiate the linear regression object
    mymodel = LinearRegression(10)

    Step 3: Load model weights from the Nillion network using the model name as
ID
    Provider provides the model and User runs inference
    mymodel.loadstatefromnetwork("mymodel", provider, na.SecretRational)

    Step 4: Load input data for inference provided by User
    myinput = na.array((10,), user, "myinput", na.SecretRational)

    Step 5: Compute inference
    result = mymodel.forward(myinput)

    Step 6: Produce the output for User with the variable name "myoutput"
    return na.output(result, user, "myoutput")
```

Now, let's break down each section one step at a time.

1. Import Section

Start by importing the necessary modules:

```
python
import nadanumpy as na
from nadaai.linearmodel import LinearRegression
```

2. Party Declaration Section

Declare the parties involved in the computation. In this case, we have a provider in charge of training the model and uploading it to the Nillion Network. The user will input its data and obtain the output classification. Both contribute to the computation but they learn nothing about each other's inputs.

```
python
    user = Party(name="User")
    provider = Party(name="Provider")
```

This line creates two parties: User and Provider.

3. Model Initialization Section

Now, let's instantiate the linear regression model. We use the Nada AI wrapper for Linear Regression and define the number of weights, matching what we initialized our model with.

```
python
    mymodel = LinearRegression(10)
```

Here, LinearRegression(10) initializes a linear regression model with 10 features. The initialization only includes the declaration of the underlying components. Next, we need to initialize the model by loading the weights. We do that with the following line:

```
python
    mymodel.loadstatefromnetwork("mymodel", provider, na.SecretRational)
```

This line loads the model weights using the model name "mymodel" for Provider. The weights are of type SecretRational.

:::tip

Under the hood, NadaArray creates variables for each element of the LinearRegression model. For example, for a model called mymodel, we will have variables mymodelcoef0 to mymodelcoef10 and mymodelintercept0.

:::

4. Input Data Loading Section

Now, we need to load the user data for the inference. We rely on Nada Numpy for this task:
python
```
    myinput = na.array((10,), user, "myinput", na.SecretRational)
```

This line creates an array myinput with 10 elements, owned by User, named "myinput", and of type na.SecretRational.

5. Inference Computation Section

With our inputs ready, we can perform computations. We use the model that we just initialized with the common forward function over the inputs.
python
```
    result = mymodel.forward(myinput)
```

This line computes the result of the linear regression inference.

6. Output Section

As a final step, we have to determine what our outputs of the computation will be. In this case, User receives myoutput as a result of the computation.
python
```
    return na.output(result, user, "myoutput")
```

This line specifies that User will receive the output named "myoutput".

With this structure in place, you can build and test your program using:
bash
```
nada build
```

Using Nada Numpy with Nillion Network

Once your program is written, you can integrate it with the Nillion Network using the Python Nada Numpy client. The process is similar to other examples. For detailed instructions and the complete code, refer to the GitHub repository.

First, import the necessary modules and the SklearnClient from Nada AI.
python
```
from sklearn.linearmodel import LinearRegression
from nadaai.client import SklearnClient
```

Then, we use SklearnClient to format our input model for upload to the Nillion Network:
python
```
Create and store model secrets via ModelClient
modelclient = SklearnClient(fitmodel)  fitmodel is the same model used for
training
modelsecrets = nillion.Secrets(
    modelclient.exportstateassecrets("mymodel", na.SecretRational)
)
```

That’s it! You’ve successfully created, built, and integrated a Nada AI Linear Regression model with the Nillion Network.

For more examples, visit our Github Repository Examples.

# nada-ai-neural-network.md:

Nada AI Multi-Layer Perceptron Tutorial

In this tutorial, we’ll explore the capabilities of Nada AI to create a simple Multi-Layer Perceptron (MLP) model for inference. First, we'll define our model, and then import it to Nada and the Nillion Network.

Before starting, make sure you have the Nillion SDK installed.

Installation

First things first, let’s get Nada Numpy installed on your system. It’s as easy as running:

```
bash
pip install --upgrade nada-ai
```

Defining our Multi-Layer Perceptron Model

If you've previously installed nada-ai, you should already have the dependencies required for this demo. We will define a simple MLP with two linear layers and a ReLU activation. Here is the script to define it:

```
python
import torch

Create custom torch Module
class MyNN(torch.nn.Module):
    """My simple neural net"""

    def init(self) -> None:
        """Model is a two layers and an activations"""
        super(MyNN, self).init()
        self.linear0 = torch.nn.Linear(8, 4)
        self.linear1 = torch.nn.Linear(4, 2)
        self.relu = torch.nn.ReLU()

    def forward(self, x: torch.tensor) -> torch.tensor:
        """My forward pass logic"""
        x = self.linear0(x)
        x = self.relu(x)
        x = self.linear1(x)
        return x

mynn = MyNN()
```

We will not be delving into the model training, as it remains out of the scope, however, we refer you to the source code in case you want to train a model with similar characteristics.

Now, let's break down the different parts of the program.

First, we import the necessary modules, in this case Pytorch with torch.

```
python
import torch
```

Next, we define our neural network class MyNN, which inherits from torch.nn.Module. The init method initializes two linear layers and a ReLU activation function.

```
python
class MyNN(torch.nn.Module):
    """My simple neural net"""

    def init(self) -> None:
        """Model is a two layers and an activations"""
        self.linear0 = nn.Linear(8, 4)
        self.linear1 = nn.Linear(4, 2)
        self.relu = nn.ReLU()
```

The forward method defines the forward pass logic of the neural network. It applies the first linear transformation, followed by a ReLU activation, and then the second linear transformation.

```
python
    def forward(self, x: na.NadaArray) -> na.NadaArray:
        """My forward pass logic"""
        x = self.linear0(x)
        x = self.relu(x)
        x = self.linear1(x)
        return x
```

Now that we've defined our model, we can proceed to migrate it to Nillion for private execution.

Writing our Nada-AI Program

To get started, initialize your project structure with the following command:

```
bash
nada init nada-mlp
```

This sets up your environment. We'll create a program that performs inference using our MLP model. Here’s the complete code to place in src/main.py:

```
python
import nadanumpy as na
from nadaai import nn

class MyNN(nn.Module):
    """My simple neural net"""

    def init(self) -> None:
        """Model is a two layers and an activations"""
        self.linear0 = nn.Linear(8, 4)
        self.linear1 = nn.Linear(4, 2)
        self.relu = nn.ReLU()

    def forward(self, x: na.NadaArray) -> na.NadaArray:
        """My forward pass logic"""
        x = self.linear0(x)
        x = self.relu(x)
        x = self.linear1(x)
        return x

def nadamain():
    Step 1: We use Nada Numpy wrapper to create "Provider" and "User"
    user = Party(name="User")
    provider = Party(name="Provider")

    Step 2: Instantiate model object
    mymodel = MyNN()

    Step 3: Load model weights from Nillion network by passing model name (acts
as ID)
    In this example, Provider provides the model and User runs inference
    mymodel.loadstatefromnetwork("mynn", provider, na.SecretRational)

    Step 4: Load input data to be used for inference (provided by User)
    myinput = na.array((8,), user, "myinput", na.SecretRational)

    Step 5: Compute inference
    Note: completely equivalent to mymodel.forward(...)
    result = mymodel(myinput)

    Step 6: We can use result.output() to produce the output for User and
variable name "myoutput"
    return result.output(user, "myoutput")
```

Now, let's break down each section one step at a time.

1. Import Section

Start by importing the necessary modules:

```
python
import nadanumpy as na
from nadaai import nn
```

2. Neural Network Definition

Define your neural network model as shown previously. Ensure this is included in src/main.py. As you can see, the model is almost a 1:1 translation from Torch to Nada AI.

<table>
<tr>
<th>Torch</th>
<th>Nada AI</th>
</tr>
<tr>
<td>

```
python
import torch

class MyNN(torch.nn.Module):
    """My simple neural net in Torch"""

    def init(self) -> None:
        """Model is a two layers and an activations"""
        super(MyNN, self).init()
        self.linear0 = torch.nn.Linear(8, 4)
        self.linear1 = torch.nn.Linear(4, 2)
        self.relu = torch.nn.ReLU()

    def forward(self, x: torch.tensor) -> torch.tensor:
        """My forward pass logic"""
        x = self.linear0(x)
        x = self.relu(x)
        x = self.linear1(x)
        return x

</td>
<td>

python
from nadaai import nn
import nadanumpy as na

class MyNN(nn.Module):
    """My simple neural net in Nada AI"""

    def init(self) -> None:
        """Model is a two layers and an activations"""
        self.linear0 = nn.Linear(8, 4)
        self.linear1 = nn.Linear(4, 2)
        self.relu = nn.ReLU()

    def forward(self, x: na.NadaArray) -> na.NadaArray:
        """My forward pass logic"""
        x = self.linear0(x)
        x = self.relu(x)
        x = self.linear1(x)
        return x

</td>
</tr>
</table>
```

3. Party Declaration Section

Declare the parties involved in the computation. In this case, we have two parties, User and Provider.

```
python
user = Party(name="User")
provider = Party(name="Provider")
```

This line creates two parties: User and Provider.

4. Model Initialization Section

Now, let's instantiate the neural network model:

```
python
mymodel = MyNN()
```

Here, MyNN() initializes our previously defined MLP model. Next, we need to load the model weights:

```
python
mymodel.loadstatefromnetwork("mynn", provider, na.SecretRational)
```

This line loads the model weights using the model name "mynn" for Provider. The weights are of type SecretRational.

5. Input Data Loading Section

Next, load the user data for inference. We rely on Nada Numpy for this task:

```python
myinput = na.array((8,), user, "myinput", na.SecretRational)
```

This line creates an array myinput with 8 elements, owned by User, named "myinput", and of type SecretRational.

6. Inference Computation Section

With our inputs ready, we can perform the computations:

```python
result = mymodel(myinput)
```

This line computes the result of the MLP inference.

7. Output Section

As a final step, we determine the output of the computation. In this case, User receives myoutput as a result of the computation.

```python
return na.output(result, user, "myoutput")
```

This line specifies that User will receive the output named "myoutput".

With this structure in place, you can build and test your program using:

```bash
nada build
```

Using Nada Numpy with Nillion Network

Once your program is written, you can integrate it with the Nillion Network using the Python Nada Numpy client. The process is similar to other examples. For detailed instructions and the complete code, refer to the GitHub repository.

First, import the necessary modules, including the TorchClient from Nada AI:

```python
from nadaai.client import TorchClient
```

Then, use TorchClient to format your input model for upload to the Nillion Network:

```python
Create and store model secrets via TorchClient
modelclient = TorchClient(mynn)
modelsecrets = nillion.Secrets(
    modelclient.exportstateassecrets("mynn", na.SecretRational)
)
```

That’s it! You’ve successfully created, built, and integrated a Nada AI MLP model with the Nillion Network.

For more examples, please visit our Github Repository Examples.

# nada-ai-reference.md:

NN Components

Module: Neural Network Module Logic

Classes

Class	Description
Module	Generic neural network module.
Flatten	Flatten layer implementation.
Linear	Linear layer implementation.
AvgPool2d	2D-Average pooling layer implementation.
DotProductSimilarity	Dot product similarity module.
ReLU	ReLU layer implementation.
Conv2d	Conv2D layer implementation.
Parameter	Parameter class.
LinearRegression	Linear regression implementation
LogisticRegression	Logistic regression implementation
Prophet	Prophet forecasting implementation

Model Clients

Class	Description
ProphetClient	ModelClient for Prophet models
SklearnClient	ModelClient for Scikit-learn models
TorchClient	ModelClient for PyTorch models
ModelClientMeta	ML model client metaclass
ModelClient	ML model client

Class: Module

Method	Description
forward(x: na.NadaArray) -> na.NadaArray	Abstract method for forward pass.
call(args, kwargs) -> na.NadaArray	Proxy for forward pass.
namedparameters(prefix: str) -> Iterator[Tuple[str, Parameter]]	Recursively generates all parameters in the module.
namedparameters() -> Iterator[Tuple[str, Parameter]]	Generates all parameters in the module.
numel() -> Iterator[int]	Recursively generates number of elements in each parameter.
numel() -> int	Returns total number of elements in the module.
loadstatefromnetwork(name: str, party: Party, nadatype: NadaInteger) -> None	Loads the model state from the Nillion network.

Class: Flatten

Method	Description
init(startdim: int = 1, enddim: int = -1) -> None	Initializes the flatten layer with start and end dimensions.
forward(x: na.NadaArray) -> na.NadaArray	Forward pass. Flattens the input tensor.

Class: Linear

Method	Description
init(infeatures: int, outfeatures: int, includebias: bool = True) -> None	Initializes the linear layer with input features, output features, and an optional bias.
forward(x: na.NadaArray) -> na.NadaArray	Forward pass. Applies a linear transformation to the input.

Class: AvgPool2d

Method	Description
init(kernelsize: ShapeLike2d, stride: Optional[ShapeLike2d] = None, padding: ShapeLike2d = 0) -> None	Initializes the 2D average pooling layer.
forward(x: na.NadaArray) -> na.NadaArray	Forward pass. Applies average pooling to the input.

Class: DotProductSimilarity

Method	Description
forward(x1: na.NadaArray, x2: na.NadaArray) -> na.NadaArray	Forward pass. Computes the dot product similarity between two input arrays.

Class: ReLU

Method	Description
forward(x: na.NadaArray) -> na.NadaArray	Forward pass. Applies the ReLU activation function to the input.
static rationalrelu(value: Union[na.Rational, na.SecretRational]) -> Union[na.Rational, na.SecretRational]	Element-wise ReLU logic for rational values.
static relu(value: NadaType) -> Union[PublicInteger, SecretInteger]	Element-wise ReLU logic for NadaType values.

Class: Conv2d

Method	Description
init(inchannels: int, outchannels: int, kernelsize: ShapeLike2d, padding: ShapeLike2d = 0, stride: ShapeLike2d = 1, includebias: bool = True) -> None	Initializes the 2D convolutional layer.
forward(x: na.NadaArray) -> na.NadaArray	Forward pass. Applies 2D convolution to the input.

Class: Parameter

Method	Description
init(shape: ShapeLike = 1) -> None	Initializes the parameter with a given shape.
numel() -> int	Returns the number of elements in the parameter.
loadstate(state: na.NadaArray) -> None	Loads a provided NadaArray as the new parameter state.

Linear Models

Class Name	Description	Methods
LinearRegression	Linear regression implementation	init(infeatures: int, includebias: bool = True) -> None<br/>forward(x: na.NadaArray) -> na.NadaArray
LogisticRegression	Logistic regression implementation	init(infeatures: int, outfeatures: int, includebias: bool = True) -> None<br/>forward(x: na.NadaArray) -> na.NadaArray

<!--

Typing Traits

Name	Description
NillionType	Union type for Nillion types including na.Rational, na.SecretRational, SecretInteger, etc.
LinearModel	Union type for linear models including LinearRegression, LogisticRegression, LogisticRegressionCV
ShapeLike	Union type for shapes including int and Sequence[int]
ShapeLike2d	Union type for 2D shapes including int and Tuple[int, int]
NadaInteger	Union type for integer types including dsl.SecretInteger, dsl.PublicInteger, etc.	-->

Time Series Models

Class Name	Description	Methods
Prophet	Prophet forecasting implementation	init(nchangepoints: int, growth: str = "linear", yearlyseasonality: bool = True, weeklyseasonality: bool = True, dailyseasonality: bool = False, seasonalitymode: str = "additive") -> None<br/> <br/> predict(dates: np.ndarray, floor: na.NadaArray, t: na.NadaArray) -> na.NadaArray<br/>predicttrend(floor: na.NadaArray, t: na.NadaArray) -> na.NadaArray<br/> <br/> predictseasonalcomps(dates: np.ndarray) -> Tuple[na.NadaArray, na.NadaArray]<br/>makeseasonalityfeatures(dates: np.ndarray, seasonalities: Dict[str, Any]) -> Dict[str, na.NadaArray]<br/> <br/> ensurenumericdates(dates: np.ndarray) -> np.ndarray<br/> <br/> call(dates: np.ndarray, floor: na.NadaArray, t: na.NadaArray) -> na.NadaArray<br/> <br/> forward(dates: np.ndarray, floor: na.NadaArray, t: na.NadaArray) -> na.NadaArray

<!-- Utility Functions

Function Name	Description	Arguments	Returns
fourierseries	Generates Fourier series	dates (np.ndarray), period (int | float), seriesorder (int)	np.ndarray
kerneloutputshape	Determines the output shape after a kernel operation	inputdims (Tuple[int, int]), padding (Tuple[int, int]), kernelsize (Tuple[int, int]), stride (Tuple[int, int])	Tuple[int, int]
ensuretuple	Ensures input gets converted to a shape tuple	tuplelike (ShapeLike2d)	Tuple[int, int]	-->

<!-- Custom Exceptions

Exception Name	Description

| MismatchedShapesException | Raised when NadaArray shapes are incompatible |
-->

Model Clients

Class Name	Description	Methods
ProphetClient	ModelClient for Prophet models	init(model: prophet.forecaster.Prophet) -> None
SklearnClient	ModelClient for Scikit-learn models	init(model: sklearn.base.BaseEstimator) -> None
TorchClient	ModelClient for PyTorch models	init(model: nn.Module) -> None
ModelClientMeta	ML model client metaclass	call(cls, args, kwargs) -> object
ModelClient	ML model client	exportstateassecrets(name: str, nadatype: NillionType) -> Dict[str, NillionType]<br/>ensurenumpy(arraylike: Any) -> np.ndarray

For more examples, please visit our Github Repository Examples.

# nada-by-example-quickstart.md:

```
---
displayedsidebar: nadaByExampleSidebar
---

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import PythonVersionInfo from './\python-version-info.mdx';
import IframeVideo from '@site/src/components/IframeVideo/index';
```

How to Run Nada by Example

Setup options

All examples for "Nada by Example" live in the nada-by-example Github repo. Nada programs are in nada-by-example/src/ and tests are in nada-by-example/tests. To run examples, there are 2 setup options:

1. Recommended - 1 Click Gitpod Setup
2. Local Machine Setup

<Tabs>

<TabItem value="gitpod" label="1 Click Gitpod Setup" default>

![Open in Gitpod](https://gitpod.io/#https://github.com/nillionnetwork/nada-by-example)

Click the button above to open the Nada by Example repo in Gitpod. Then follow the run and test an example section to learn how to run and test a Nada program.

<IframeVideo

videoSrc="https://www.loom.com/embed/4395eeed66934142ba0feaf68a43534a?sid=785f1cc2-881a-4c30-8d96-07007251bd6b"/>

</TabItem>

<TabItem value="local" label="Local Machine Setup">
Complete local repo setup following the instructions below to run an example from the repo locally.

Install Nillion, clone the nada-by-example repo, and set up a developer environment for your local nada-by-example repo.

1. Install Nillion globally

Check to see if you have nilup, the Nillion installer installed

```
nilup -V
```

If you don't have nilup, install nilup

```
curl https://nilup.nilogy.xyz/install.sh | bash
```

2. Use the latest version of the Nillion SDK

Install and use the latest version of the Nillion SDK and tools.

```
nilup install latest
nilup use latest
```

3. Optionally enable nilup telemetry, providing your Ethereum wallet address

```
nilup instrumentation enable --wallet <your-eth-wallet-address>
```

4. Clone the nada-by-example repo

Star nada-by-example on Github so you have it for future reference. Then clone the repo

```
git clone https://github.com/NillionNetwork/nada-by-example.git
```

5. Create a Python virtual environment and install Nada in nada-by-example

<PythonVersionInfo/>

```
cd nada-by-example
python3 -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
```

6. Build (compile) all Nada programs in the repo

```
nada build
```

nada build is a nada tool command that compiles programs to create one compiled binary (.nada.bin) file per program listed in the nada-project.toml file in the target/ directory.

✅ Great work! Now that you've set up the nada-by-example repo, you can run any example.
</TabItem>
</Tabs>

How to run and test an example program

:::tip

The nada-by-example repo is a Nada project created with Nillion's nada tool. You can run all existing nada commands found in the nada tool docs to compile programs, get program requirements, run a program, and test a program.
:::

Every Nada program example has a corresponding test file. Programs are in nada-by-example/src/ and test files are in nada-by-example/tests. Running a program uses the inputs specified in the test file. Testing a program uses the inputs specified in the test file and also checks the outputs against the expectedoutputs specified in the test file.

Run any program with the inputs specified in the test file with nada run

```
nada run <test-file-name>
```

Test any program with the inputs and outputs specified in the test file with nada test

```
nada test <test-file-name>
```

Run the addition example

Here is the Nada program and test file for the addition example. The program is src/addition.py and the test file is tests/additiontest.yaml

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/addition.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/additiontest.yaml

</TabItem>
</Tabs>

Run the addition program with additiontest test inputs:

```
nada run additiontest
```

The result of running this program is

```
(.venv) ➜  nada-by-example git:(main) nada run additiontest
Running program 'addition' with inputs from test file additiontest
Building ...
Running ...
Program ran!
Outputs: {
    "sum": SecretInteger(
        NadaInt(
            40,
        ),
    ),
}
```

Test the addition example

Here is the Nada program and test file for the addition example. The program is src/addition.py and the test file is tests/additiontest.yaml

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/addition.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/additiontest.yaml

</TabItem>
</Tabs>

Test the addition program with additiontest test inputs and expected outputs:

```
nada test additiontest
```

The result of testing this program is

```
(.venv) ➜  nada-by-example git:(main) nada test additiontest
Running test: additiontest
Building ...
Running ...
additiontest: PASS
```

Testing the addition program with additiontest results in a PASS because the expectedoutputs sum output matches the run result.

Add a new test for the addition example

Use the nada tool to add a new test file named "additiontest2" for the addition example.

```
nada generate-test --test-name additiontest2 addition
```

This results in a new test file: /tests/additiontest2.yaml

```
(.venv) ➜  nada-by-example git:(main) nada generate-test --test-name
additiontest2 addition
Generating test 'additiontest2' for
Building ...
Generating test file ...
Test generated!
```

Update the values in the test file to anything you want, for example:

```yaml
---
program: addition
inputs:
  num1:
    SecretInteger: '100'
  num2:
    SecretInteger: '10'
expectedoutputs:
  sum:
    SecretInteger: '110'
```

Run addition with your new test

```
nada run additiontest2
```

Test addition with your new test

```
nada test additiontest2
```

Keep exploring examples

🥳 You're all set up to run and test any example in nada-by-example. Keep exploring what's possible with Nada by running the rest of the programs in the repo.

# nada-by-example.md:

---
displayedsidebar: nadaByExampleSidebar

Nada by Example

Welcome to Nada by Example, an introduction to the Nada language with reference examples from the nada-by-example repo. Nada DSL is a Python-based language for defining MPC (Multi-Party Compute) programs on the Nillion Network.

Developers can use Nada to write programs that handle secret inputs from multiple parties (i.e., multiple users). These programs compute in a way that ensures they are “blind” to the underlying data, meaning they cannot access or view the secret inputs. This concept is known as “blind computation.”

To learn more about Nada, check out the full Nada language docs here.

# nada-debugging.md:

Debugging Nada

Nada DSL provides a way to easily interface with MPC in a user-friendly programming language. However, given the restrictions of MPC, behaviors that Python as the metaprogramming layer would enable are often not permitted. When trying to debug programs, there are a few strategies that can help us understand what's happening in Nada.

Think Big, Code Small

A fundamental part of successfully developing complex functionality in Nada relies on following a set of best practices when developing your programs. Debugging large pieces of code can be a nightmare. Thus, setting up your project in an appropriate manner can help avoid headaches. Our recommendation here is to build and test small pieces of code. For that, we can use the nada project management tool and its capabilities.

These few tips when setting up your project can help ensure a healthier development lifecycle:

1. Don't use main to write your code: Instead, write it in another .py file, import it, and execute it. This way, we can use multiple test files for each piece of code, ensuring consistent behavior.

```
   <table>
   <tr>
   <th>DO:</th>
   <th>DON'T:</th>
   </tr>
   <tr>
   <td>

   bash
   src
   ├── lib.py
   └── main.py


   python
   lib.py
   from nadadsl import

   def func(a, b):
       return a + b


   python
   main.py
   from nadadsl import
   from lib import func

   def main():
       ...
       c = func(a, b)
       ...
   </td>
   <td>

   bash
   src
   └── main.py


   python
   main.py

   from nadadsl import

   def func(a, b):
       return a + b

   def main():
       ...
       c = func(a, b)
       ...


   </td>
   </tr>
   </table>
```

2. Create and test multiple compilation units: Using the nada project management tool, one can create multiple independent compilation units, that is, programs that compile independently. To create a separate compilation unit, follow these steps:

   - Add the following line to nada-project.toml:

```
   toml
   [[programs]]
   path = "src/testfunc.py" testfunc must be replaced by the name of your
choice.
   primesize = 128 Can also be 64 or 256
```

   - Create the appropriate file in the src/ directory, in this case, testfunc.py:

```
   bash
   src
   ├── lib.py
   ├── main.py
   └── testfunc.py
```

   - Use this file to test a single functionality, in this case our function func:

```
   python
   testfunc.py
   from nadadsl import
   from lib import func

   def main():
       ...
       c = func(a, b)
       ...
```

After this point, whenever we execute nada build, nada will compile all the programs in the nada-project.toml file. This can serve as a way to test that any changes do not break other functionalities. Furthermore, if you wish to compile a single file, you can do so with: nada build <file>, in this case nada build testfunc.

3. Add Tests for Every Function: Adding tests for every function can be a good way to detect wrong behavior and to test for corner cases. For that, you can use nada generate-test to produce a test file named testfunc1.yaml in the tests/ directory, with the following command:

```
bash
nada generate-test --test-name <testname> <sourcefile>
nada generate-test --test-name testfunc1 testfunc
```

This will produce the following tree:

```
bash
├── src
│   ├── lib.py
│   ├── main.py
│   └── testfunc.py
└── tests
    └── testfunc1.yaml
```

You can create as many tests as you wish for the same compilation unit. If you execute nada test, it will execute all the existing tests for all compilation units. This can be a good way to check for any regression errors in your Nada programs. If you wish to execute the tests for a single file, you can do so with nada test testfunc1.

4. Use print to debug: Follow our Debugging with print() guide to learn how to inspect the values and types of variables at various points in your Nada program code.

# nada-lang-operators.md:

import NadaOperationsTable from './\operations-table.mdx';

Built-In Operations

Overview of the primitive and the array Nada operations.

Primitive Operations

<NadaOperationsTable/>

# nada-lang-programs.md:

Programming with Nada

This is a basic introduction to programming with the Nada embedded domain-specific language (DSL). Use this page to learn how to compile and run a basic Nada program. Once you are ready, proceed to the other tutorials to learn Nada features via simple example programs!

Example Programs

Tiny Addition (with a Single Party)

Below is a tiny Nada program tinyaddition.py that adds two secret integer inputs. Both of these integer inputs belong to Nilla 🐶, who is using the Nillion Network to compute their total. Most importantly, the secret integers are never revealed to the Nillion Network nodes!

```
python
from nadadsl import

def nadamain():
    nillathedog = Party(name="Nilla 🐶")

    secretint1 = SecretInteger(Input(name="mysecret1", party=nillathedog))
    secretint2 = SecretInteger(Input(name="mysecret2", party=nillathedog))

    total = secretint1 + secretint2

    return [Output(total, "myoutput", nillathedog)]
```

The tinyaddition.py program takes in two Inputs from the same Party.

```
<table><thead><tr><th width="162">Input name</th><th width="145">Input
type</th><th>Party
name</th></tr></thead><tbody><tr><td>mysecret1</td><td>SecretInteger</
td><td>Nilla
🐶</td></tr><tr><td>mysecret2</td><td>SecretInteger</td><td>Nilla
🐶</td></tr></tbody></table>
```

The program tinyaddition.py returns an Output to a Party. Only that party sees the output because it it of type SecretInteger.

```
<table>
  <thead>
    <tr><th>Output value</th><th>Output Name</th><th>Output Type</th><th>Party
name</th></tr>
  </thead>
  <tbody>
    <tr><td>total</td><td>myoutput</td><td>SecretInteger</td><td>Nilla
🐶</td></tr>
  </tbody>
</table>
```

Ternary conditional operators

Next, let's see an example of our ternary operation:

```
python reference showGithubLink
https://github.com/nillion-oss/nillion-python-starter/blob/main/programs/
nadafnmax.py
```

Multiple parties in Nada

Lastly, below is an example with two parties:

```
python reference showGithubLink
https://github.com/NillionNetwork/python-examples/blob/main/
examplesandtutorials/nadaprograms/src/additionsimplemultiparty.py
```

For more examples visit our programs directory.

Compile and run Nada programs

The nada tool can be used to manage Nada projects (create project, compile, run,

and test programs, generate tests, etc.).
Alternatively, you can use the pynadac and nada-run standalone tools.

Compile a Nada program with pynadac

Before the programs can be stored and run on the Nillion Network, they must be compiled into bytecode.
The Nada Compiler, referred to as pynadac, is a developer tool within the Nillion SDK used to compile programs written in Nada to bytecode.

Compiling the example above outputs the compiled circuit to bytecode in a new file:

- Input program: tinyaddition.py
- Output bytecode: tinyaddition.nada.bin

Run a program locally with the nada-run

Use the nada-run developer tool within the Nillion SDK to quickly execute and iterate on Nada programs under an environment that is a close as it can get to running them in a real network. The local network tool:

- takes in a compiled Nada program and secrets,
- creates blinding factors locally to hide every secret input you provide to the program,
- creates a stripped down version of a Nillion cluster and runs the same bytecode that would be run during a real execution of your program.&#x20;

```
python
secrets file
integers:
  mysecret1: 6
  mysecret2: 4
```

Local program execution goes through the same flows your code would on the Nillion Network. Here's how you could execute the program locally from the command line assuming you had the following files

- secrets
- tinyaddition.nada.bin

```
./nada-run --secrets-path ./secrets tinyaddition.nada.bin
```

The result of executing the tinyaddition.nada.bin program on the secret inputs from the secrets file with nada-run is 10.

# nada-lang-tutorial-arithmetic-and-logic.md:

Arithmetic and Logic

Nada programs can work with common arithmetic and logic values (such as integers and booleans) and operations (such as addition and conditional expressions).

Integers

The example below computes the revenue generated from the sales of two categories of product. In this example, all inputs are of type SecretInteger, so the result revenue is also of this type.

<iframe src='img/nada-lang-tutorial-arithmetic-and-logic-0.html' height='350px'

```
width='100%'></iframe>
<!--python
from nadadsl import

def nadamain():
pricing = Party(name="pricing")
inventory = Party(name="inventory")
accounting = Party(name="accounting")

    pricepotato = SecretInteger(Input(name="pricepotato", party=pricing))
    pricetomato = SecretInteger(Input(name="pricetomato", party=pricing))

    quantitypotato = SecretInteger(Input(name="quantitypotato",
party=inventory))
    quantitytomato = SecretInteger(Input(name="quantitytomato",
party=inventory))

    revenue = (pricepotato  quantitypotato) + (pricetomato + quantitytomato)

    return [Output(revenue, "revenue", accounting)]

-->
```

Suppose that the price information is public. In this case, revenue is still of type SecretInteger because the quantity information is private. If revenue were of type PublicInteger, it would (in some cases) be possible to determine the quantity information from the revenue by working backwards.

```
<iframe src='img/nada-lang-tutorial-arithmetic-and-logic-1.html' height='350px'
width='100%'></iframe>
<!--python
from nadadsl import

def nadamain():
    pricing = Party(name="pricing")
    inventory = Party(name="inventory")
    accounting = Party(name="accounting")

    pricepotato = PublicInteger(Input(name="pricepotato", party=pricing))
    pricetomato = PublicInteger(Input(name="pricetomato", party=pricing))

    quantitypotato = SecretInteger(Input(name="quantitypotato",
party=inventory))
    quantitytomato = SecretInteger(Input(name="quantitytomato",
party=inventory))

    revenue = (pricepotato  quantitypotato) + (pricetomato + quantitytomato)

    return [Output(revenue, "revenue", accounting)]
-->
```

What about integer values that appear in the program as literals (i.e., they are not secret or public inputs) but are used within calculations involving inputs? These should be of type Integer.

```
<iframe src='img/nada-lang-tutorial-arithmetic-and-logic-2.html' height='300px'
width='100%'></iframe>
<!--python
from nadadsl import

def nadamain():
    pricing = Party(name="pricing")
    inventory = Party(name="inventory")
    accounting = Party(name="accounting")
    price = PublicInteger(Input(name="price", party=pricing))
    quantity = SecretInteger(Input(name="quantity", party=inventory))

    revenueincents = Integer(100)  price  quantity

    return [Output(revenueincents, "revenueincents", accounting)]
-->
```

Boolean Values and Comparison of Integers

Comparison operations can be applied to integers. Such comparison expressions evaluate to Nada boolean values. Whether this resulting value is secret depends on whether the integers being compared are secret. Furthermore, Nada boolean values support the ifelse method, which implements a variant of the ternary conditional operator that can work with Nada values (even if they are secret).

The example below leverages both an integer comparison operator and the ternary operator to determine the larger of two secret inputs.

```
<iframe src='img/nada-lang-tutorial-arithmetic-and-logic-3.html' height='278px' width='100%'></iframe>
<!--python
from nadadsl import

def nadamain():
    dataowner = Party(name="dataowner")

    x = SecretInteger(Input(name="x", party=dataowner))
    y = SecretInteger(Input(name="y", party=dataowner))

    condition = x > y
    maximum = condition.ifelse(x, y)

    return [Output(maximum, "maximum", dataowner)]
-->
```

Built-in Python Constants and Operations

Because Nada is a DSL embedded inside Python, built-in constants of type int and bool (and the operators associated with these types) can be used directly. However, it is important to understand that these cannot be used interchangeably or mixed. The example below demonstrates both correct and incorrect usage of built-in and Nada values.

```
<iframe src='img/nada-lang-tutorial-arithmetic-and-logic-4.html' height='350px' width='100%'></iframe>
<!--python
from nadadsl import

def nadamain():
    dataowner = Party(name="dataowner")

    x = SecretInteger(Input(name="x", party=dataowner))

    Permitted.
    a = x + Integer(123) + Integer(456)
    b = x + Integer(123 + 456)

    Not permitted.
    c = x + 123 + 456
    d = x + 123

    return [Output(a, "a", dataowner), Output(b, "b", dataowner)]
```

```
-->
```

# nada-lang-tutorial-functions.md:

Functions

The Nada DSL supports a limited form of user-defined functions.

Basic Example

The example below introduces a function that calculates the total of three secret integer values. Notice that the arguments and the function itself both have Python type annotations.

```
<iframe src='img/nada-lang-tutorial-functions-0.html' height='334px'
width='100%'></iframe>
<!--python
from nadadsl import

def total(x: SecretInteger, y: SecretInteger, z: SecretInteger) ->
SecretInteger:
    return x + y + z

def nadamain():
    dataowner = Party(name="dataowner")

    x = SecretInteger(Input(name="x", party=dataowner))
    y = SecretInteger(Input(name="y", party=dataowner))
    z = SecretInteger(Input(name="z", party=dataowner))

    t = total(x, y, z)

    return [Output(t, "t", dataowner)]
-->
```

# nada-lang-tutorial-lists-and-comprehensions.md:

Lists and Comprehensions

The Nada DSL supports the introduction and use of Python lists that contain secret integer values. This includes using a subset of the list comprehension syntax supported by Python.

Basic Example

The program below uses Python list comprehensions to build an ascending sequence of three secret integers in which the first entry is a secret integer input. This sequence of secret integers is then returned by the program.

```
<iframe src='img/nada-lang-tutorial-lists-and-comprehensions-0.html'
height='278px' width='100%'></iframe>
<!--python
from nadadsl import

def nadamain():
    dataowner = Party(name="dataowner")

    start = SecretInteger(Input(name="start", party=dataowner))

    sequence = [start + Integer(i) for i in range(3)]
    return [
        Output(sequence[i], "sequence" + str(i), dataowner)
        for i in range(3)
    ]
-->
```

Voting Example using List Comprehensions

The program below assembles the secret votes from four voting parties (i.e., voters) and returns the total for each of the two candidates. Because each voting party submits an input of either 1 or 2 for each candidate, the value Integer(4) is subtracted from the total for each candidate.

```
<iframe src='img/nada-lang-tutorial-lists-and-comprehensions-1.html'
height='640px' width='100%'></iframe>
<!--python
from nadadsl import

def total(xs: list[SecretInteger]) -> SecretInteger:
    return xs[0] + xs[1] + xs[2] + xs[3]

def nadamain():
    Create the voter parties and the voting official party.
    voters = [Party("voter" + str(v)) for v in range(4)]
    official = Party(name="official")

    Gather the inputs (one vote for each candidate from each voter).
    votespercandidate = [
        [
            SecretInteger(
                Input(
                    name="voter" + str(v) + "candidate" + str(c),
                    party=Party("voter" + str(v))
                )
            )
            for v in range(4)
        ]
        for c in range(2)
    ]

    Calculate and return the total for each candidate.
    return [
        Output(
            total(votespercandidate[c]) - Integer(4),
            "candidate" + str(c),
            official
        )
        for c in range(2)
    ]
-->
```

A list comprehension is used to construct the list of parties corresponding to the voters. An expression containing a Python list comprehension nested in another list comprehension is used to assemble a list of lists votespercandidate that contains two lists (i.e., a list of the votes submitted for each of the two candidates). Finally, a list comprehension is used to build the list of outputs.

# nada-lang-tutorial-lists-and-iteration.md:

Lists and Iteration

The Nada DSL supports the use of iteration over lists that contain secret integer values. This includes limited use of for loops.

Basic Example

The program below uses a Python for loop to build an ascending sequence of three secret integers in which the first entry is a secret integer input. This sequence of secret integers is then returned by the program. Notice that when the lists sequence and outputs are first defined, their types are explicitly specified using a Python type annotation.

```
<iframe src='img/nada-lang-tutorial-lists-and-iteration-0.html' height='422px'
width='100%'></iframe>
<!--python
from nadadsl import

def nadamain():
    dataowner = Party(name="dataowner")

    start = SecretInteger(Input(name="start", party=dataowner))

    sequence: list[SecretInteger] = []
    for i in range(3):
        sequence.append(start + Integer(i))

    outputs: list[Output] = []
    for i in range(3):
        outputs.append(Output(
            sequence[i],
            "sequence" + str(i),
            dataowner
        ))

    return outputs
-->
```

Voting Example using Iteration over Lists

The program below assembles the secret votes from four voting parties (i.e., voters) and returns the total for each of the two candidates. Because each voting party submits an input of either 1 or 2 for candidate, the value Integer(4) is subtracted from the total for each candidate.

```
<iframe src='img/nada-lang-tutorial-lists-and-iteration-1.html' height='730px'
width='100%'></iframe>
<!--python
from nadadsl import

def total(xs: list[SecretInteger]) -> SecretInteger:
    return xs[0] + xs[1] + xs[2] + xs[3]

def nadamain():
    Create the voter parties and the voting official party.
    voters: list[Party] = []
    for v in range(4):
        voters.append(Party("voter" + str(v)))
    official = Party(name="official")

    Gather the inputs (one vote for each candidate from each voter).
    votespercandidate: list[list[SecretInteger]] = []
    for c in range(2):
        votespercandidate.append([])
        for v in range(4):
            votespercandidate[c].append(SecretInteger(
                Input(
                    name="voter" + str(v) + "candidate" + str(c),
                    party=Party("voter" + str(v))
                )
            ))

    Calculate and return the total for each candidate.
    Calculate the total for each candidate.
    outputs: list[Output] = []
    for c in range(2):
        outputs.append(
            Output(
                total(votespercandidate[c]) - Integer(4),
                "candidate" + str(c),
                official
            )
        )

    return outputs
-->
```

A for loop is used to build up the list of parties corresponding to the voters using the append method for lists. A for loop nested inside another for loop is used to assemble a list of lists votespercandidate that contains two lists (i.e., a list of the votes submitted for the first candidate and a list of the votes submitted for the second candidates). Finally, the list of outputs is assembled using a for loop and returned.

# nada-lang-types.md:

import NadaDataTypesTable from './\data-types-table.mdx';

Data Types

Overview of the primitive and the compound Nada data types.

Primitive Data Types

<NadaDataTypesTable/>

Secret and Public data types can be used to specify user inputs as:

```
python
a = SecretInteger(Input(name="a", party=party1))
b = SecretUnsignedInteger(Input(name="b", party=party2))
c = PublicInteger(Input(name="c", party=party1))
d = PublicUnsignedInteger(Input(name="d", party=party2))
e = SecretBoolean(Input(name="e", party=party1))
f = PublicBoolean(Input(name="f", party=party2))
```

Similarly, Literals can only be used within a program as:

```
python
a = SecretInteger(Input(name="a", party=party1))
b = SecretUnsignedInteger(Input(name="b", party=party2))
c = SecretBoolean(Input(name="c", party=party1))
newint = a + Integer(13)
newuint = b + UnsignedInteger(13)
newbool = Boolean(True)
```

# nada-lang.md:

---
description: 'Language components: Nada, the Nada compiler, and a Nada programs'
---

import DocCardList from '@theme/DocCardList';

Nada Language

Overview

The Nillion Network leverages Nada, our MPC language, for defining MPC programs. Our initial implementation of Nada comes in the form of Nada, a Python DSL (Domain Specific Language).

The Nada language is:
- Strongly typed: every variable and expression has a specific type that is checked at compile time to prevent common errors such as type mismatches.
- Correctness-oriented: Nada has features including type-checking and static analysis built into the compiler.
- Compiled: Nada is a compiled language with different stages.

Nada Language Components

<DocCardList />

# nada-libraries.md:

import DocCardList from '@theme/DocCardList';

Nada Libraries

The Nillion Network leverages Nada for defining programs. We are developing a set of libraries to enhance the developer experience when developing with Artificial Intelligence models and beyond on Nillion. These libraries are built on top of Nada DSL and are meant to be intercompatible with it.

Nada Numpy

nada-numpy is a Python library for algebraic operations on NumPy-like arrays using Nada DSL and the Nillion Network. It offers an easy-to-use interface for various computations like dot products, element-wise operations, and stacking, with broadcasting support similar to NumPy. Key Features include

- Matrix operations: Perform common matrix operations on NadaArrays (a NumPy-like matrix object).
- Element-wise Operations: Perform matrix arithmetic with support for broadcasting.
- Rational Number Support: Use Rational and SecretRational for simplified decimal number operations on Nillion.

Nada AI

nada-ai is a Python library for performing ML/AI tasks using Nada DSL and the Nillion Network. It offers an intuitive interface and seamless integration with ML frameworks like PyTorch and Sci-kit learn. Key Features include

- Exporting Model State: Integrates with models from existing ML frameworks, making it easy to export them to the Nillion network for use in Nada programs.
- AI Modules: Provides a PyTorch-like interface to create ML models in Nada by stacking pre-built common components, with the option to create custom components.
- Importing Model State: Easily import an exported model state from the Nillion

network for use in a Nada program.

<DocCardList />

# nada-numpy-array-functions.md:

Nada Array Functions

Function	Signature	Description
array[i]	item	Retrieve an item from the array.
array[i] = ...	key, value	Set an item in the array.
add	other: Any	Perform element-wise addition with broadcasting.
sub	other: Any	Perform element-wise subtraction with broadcasting.
mul	other: Any	Perform element-wise multiplication with broadcasting.
divide	other: Any	Perform element-wise division with broadcasting.
matmul	other: NadaArray	Perform matrix multiplication with another NadaArray.
dot	other: NadaArray	Compute the dot product between two NadaArray objects.
hstack	other: NadaArray	Horizontally stack two NadaArray objects.
vstack	other: NadaArray	Vertically stack two NadaArray objects.
reveal		Reveal the elements of the array.
apply	func: Callable[[Any], Any]	Apply a Python function element-wise to the array.
mean	axis=None, dtype=None, out=None	Compute the mean along the specified axis.
output	party: Party, prefix: str	Generate a list of Output objects for each element in the NadaArray.
array	dims: Sequence[int], party: Party, prefix: str, nadatype: type	Create a NadaArray with specified dimensions and element type.
random	dims: Sequence[int], nadatype: type = SecretInteger	Create a random NadaArray with specified dimensions and element type.
len		Get the length of the NadaArray.
empty		Check if the NadaArray is empty.
dtype		Get the data type of the NadaArray.
isrational		Check if the NadaArray contains rationals.
isinteger		Check if the NadaArray contains signed integers.
isunsignedinteger		Check if the NadaArray contains unsigned integers.
isboolean		Check if the NadaArray contains booleans.
str(array)		Get a string representation of the NadaArray.
debug	array: np.ndarray	Get a debug representation of the NadaArray.

# nada-numpy-dot-product.md:

Nada Numpy Dot Product Tutorial

In this tutorial, we’ll explore the capabilities of Nada Numpy together. Before we start, make sure you have the Nillion SDK installed.

To get started, we can initialize our project structure with the following command:
bash
```
nada init nada-dot-product
```

This sets up everything we need. Now, let’s dive into the code! We’ll create a program that computes the dot product of two arrays. Here’s the complete code to put in src/main.py:

python
```
from nadadsl import

Step 0: Nada Numpy is imported with this line
import nadanumpy as na


def nadamain():
    Step 1: We use Nada Numpy wrapper to create "Party0", "Party1" and "Party2"
    parties = na.parties(3)

    Step 2: Party0 creates an array of dimension (3, ) with name "A"
    a = na.array([3], parties[0], "A", SecretInteger)

    Step 3: Party1 creates an array of dimension (3, ) with name "B"
    b = na.array([3], parties[1], "B", SecretInteger)

    Step 4: The result is of computing the dot product between the two
    result = a.dot(b)

    Step 5: We can use result.output() to produce the output for Party2 and
variable name "myoutput"
    return na.output(result, parties[1], "myoutput")
```

Let’s break this down step-by-step.

1. Import Section

Start by importing the necessary modules. We need Nada DSL and Nada Numpy, so we import them like this:

python
```
from nadadsl import Output, SecretInteger
import nadanumpy as na
```

Next, define the main function for our program:

python
```
def nadamain():
```

2. Party Declaration Section

In this section, we’ll declare the parties involved in our computation. Parties represent the different participants in our Nada program. Nada Numpy makes it easy to create multiple parties with the na.parties function:

python
```
    parties = na.parties(3)
```

This line creates a list of three parties named: Party0, Party1, and Party2.

3. Input Declaration Section

Now, let’s define our inputs. We’ll create two NadaArrays, each with three elements. The na.array function allows us to define an array with arbitrary dimensions and types.

python
```
    a = na.array([3], parties[0], "A", SecretInteger)
```

This line creates an array a with 3 elements, owned by Party0, named “A”, and of type SecretInteger.

Similarly, we define the second array b:

python
```
    b = na.array([3], parties[1], "B", SecretInteger)
```

This array is also 3 elements long, owned by Party1, named “B”, and of type SecretInteger.

:::tip
Under the hood, NadaArray creates variables for each element of the array. For example, a will have variables "A0", "A1", and "A2", while b will have "B0", "B1", and "B2".
:::

4. Computation Section

With our inputs ready, we can perform computations. Calculating the dot product of the two arrays is straightforward. In the Numpy fashion, we use the dot method:

python
```
    result = a.dot(b)
```

This line computes the dot product of arrays a and b. If our input arrays were a = [1, 2, 3] and b = [4, 5, 6], the dot product would be computed as 14 + 25 + 36 = 32.

5. Output Section

Finally, we need to produce the output. Since NadaArray is not a base Nada type, we use the .output method to generate the output. This method takes the output party and the output variable name as arguments:

python
```
    return na.output(result, parties[2], "myoutput")
```

In this case, we invoke na.output(result, parties[2], "myoutput") to specify that the output party will be Party2 and the name of the output variable will be "myoutput".

With everything in place, we can build and test our program:

```bash
nada build
```

Using Nada Numpy with Nillion Network

After completing the program writing, we can upload it and interact with it in the network with the ease provided by Nada Numpy. For that, we use the Python Nillion Client. We can use the same complete code as for other examples. The only difference is how Nada Numpy allows to easily include arrays in our uploads to the Nillion Network with the Nada Numpy client. We add the link to the complete code.

First, import the necessary modules:

```python
import numpy as np
import nadanumpy.client as naclient
```

Then, we can use it to introduce arrays with a similar syntax that will take care of the array formatting naclient.array. The first element will be the array with the secret values that we want to upload. The second value is the name, which shall match with the initial name set on the Nada Program.

```python
A = np.ones((3,)) Sample numpy array with ones [1, 1, 1]
storedsecret = nillion.Secrets(naclient.array(A, "A"))
```

And that’s it! You’ve successfully created, built, and integrated a Nada Numpy program with the Nillion Network.

For more examples, please visit our Github Repository Examples.

# nada-numpy-introduction.md:

Nada Numpy

nada-numpy is an opinionated clone of Numpy, seamlessly integrated with the Nada DSL. It allows for fast and efficient manipulation of array structures using Nada DSL, while extending functionality to include interfacing with decimal numbers.

One key difference between Numpy and Nada Numpy is that Nada Numpy enforces strong typing. Unlike Numpy, which dynamically changes types, Nada Numpy requires a strongly-typed interface, working with root families (Integer, UnsignedInteger, Rational, and SecretBoolean).

In this tutorial, we’ll explore the capabilities of Nada Numpy together. Before we start, make sure you have the Nillion SDK installed.

Installation

First things first, let’s get Nada Numpy installed on your system. It’s as easy as running:

```bash
pip install --upgrade nada-numpy
```

Writing a Nada-Numpy program

Nada Numpy aligns with the structure of the Nada DSL. A typical Nada Numpy program comprises the following sections:

1. Import Section
2. Party Declaration Section
3. Input Declaration Section
4. Computation Section
5. Output Section

To get started, we can initialize our project structure with the following command:

```
bash
nada init hello-nada-numpy
```

This sets up everything we need. Now, let’s dive into the code! We’ll create a program that adds two arrays, similar to the Nada DSL starter which adds two variables. Here’s the complete code to put in src/main.py:

```
python
from nadadsl import
import nadanumpy as na

def nadamain():
    parties = na.parties(3)

    a = na.array([3], parties[0], "A", SecretInteger)
    b = na.array([3], parties[1], "B", SecretInteger)

    result = a + b

    return result.output(parties[2], "myoutput")
```

Let’s break this down step-by-step.

1. Party Declaration Section

Start by importing the necessary modules. We need Nada DSL and Nada Numpy, so we import them like this:

```
python
from nadadsl import
import nadanumpy as na
```

Next, define the main function for our program:

```
python
def nadamain():
```

In this section, we’ll declare the parties involved in our computation. Parties represent the different participants in our Nada program. Nada Numpy makes it easy to create multiple parties with the na.parties function:

```
python
    parties = na.parties(3)
```

Instead of having to define each party manually, we use Nada Numpy functionality. In this case with na.parties(3) we'll be creating a list of three parties named: Party0, Party1 and Party2.

2. Input Declaration Section

Now, let’s define our inputs. We’ll create two NadaArrays, each with three elements. The na.array function allows us to define an array with arbitrary dimensions and types.

```
python
    a = na.array([3], parties[0], "A", SecretInteger)
```

This line creates an array a with 3 elements, owned by Party0, named “A”, and of type SecretInteger.

Similarly, we define the second array b:

```
python
    b = na.array([3], parties[1], "B", SecretInteger)
```

This array is also 3 elements long, owned by Party1, named “B”, and of type SecretInteger.

:::tip

Under the hood, NadaArray creates variables for each element of the array. For example, a will have variables "A0", "A1", and "A2", while b will have "B0", "B1", and "B2".

:::

3. Computation Phase

With our inputs ready, we can perform computations. Adding the two arrays is straightforward. In the Numpy fashion, we'll be operating all the variables consecutively with the + operator.

```
python
    res = a + b
```

This line operates each entry of the array linearly. If our input arrays were: a = [1, 2, 3] and b = [4, 5, 6] we would be summing each entry of array a with the corresponding entry of array b. That is: res[0] = a[0] + b[0], res[1] = a[1] + b[1] and res[2] = a[2] + b[2]; and obtaining res = [5, 7, 9].

:::tip

You can experiment with different operations (+, -, \, /) and array dimensions to see what Nada Numpy can do!

:::

4. Output Section

Finally, we need to produce the output. Since NadaArray is not a base Nada type, we use the .output method to generate the output. This method takes the output party and the output variable name as arguments:

```
python
    return res.output(parties[2], "myoutput")
```

In this case, we'll be invoking res.output(parties[2], "myoutput") establishing that the output party will be Party2and the name of the output variable will be "myoutput".

With everything in place, we can build and test our program:

```
bash
nada build
```

Using Nada Numpy with Nillion Network

After completing the program writing, we can upload it and interact with it in the network with the ease provided by Nada Numpy. For that, we use the Python Nillion Client. We can use the same complete code as for other examples. The only difference is how Nada Numpy allows to easily include arrays in our uploads to the Nillion Networ with the Nada Numpy client. We add the link to the complete code.

First, import the necessary modules:

```
python
import numpy as np
import nadanumpy.client as naclient
```

Then, we can use it to introduce arrays with a similar syntax that will take care of the array formatting naclient.array. The first element will be the array with the secret values that we want to upload. The second value is the name, which shall match with the initial name set on the Nada Program.

```
python
A = np.ones((3,)) Sample numpy array with ones [1, 1, 1]
storedsecret = nillion.Secrets(naclient.array(A, "A"))
```

And that’s it! You’ve successfully created, built, and integrated a Nada Numpy program with the Nillion Network.

For more examples, please visit our Github Repository Examples.

# nada-numpy-matrix-multiplication.md:

Nada Numpy Matrix Multiplication Tutorial

In this tutorial, we’ll delve into the capabilities of Nada Numpy with a focus on matrix multiplication using the @ operator. Before we begin, ensure you have the Nillion SDK installed.

To get started, initialize your project structure with the following command:

```
bash
nada init nada-matmul
```

This sets up your environment. Now, let's dive into the code! We'll create a program that computes the matrix product of two arrays. Here’s the complete code to place in src/main.py:

```
python
from nadadsl import

Step 0: Nada Numpy is imported with this line
import nadanumpy as na
```

```
def nadamain():
    Step 1: We use Nada Numpy wrapper to create "Party0", "Party1" and "Party2"
    parties = na.parties(3)

    Step 2: Party0 creates an array of dimension (3 x 3) with name "A"
    a = na.array([3, 3], parties[0], "A", SecretInteger)

    Step 3: Party1 creates an array of dimension (3 x 3) with name "B"
    b = na.array([3, 3], parties[1], "B", SecretInteger)

    Step 4: The result is of computing the dot product between the two which is
another (3 x 3) matrix
    result = a @ b

    Step 5: We can use result.output() to produce the output for Party2 and
variable name "myoutput"
    return result.output(parties[1], "myoutput")
```

Now, let's break down each section step-by-step.

1. Import Section

Start by importing the necessary modules:

```python
from nadadsl import Output, SecretInteger
import nadanumpy as na
```

2. Party Declaration Section

Declare the parties involved in the computation using na.parties:

```python
    parties = na.parties(3)
```

This line creates three parties: Party0, Party1, and Party2.

3. Input Declaration Section

Define the input matrices using na.array:

```python
    A = na.array([3, 2], parties[0], "A", SecretInteger)
```

Here, A is a matrix of size 3x2, owned by Party0, named "A", and of type SecretInteger.

Similarly, define matrix B:

```python
    B = na.array([2, 4], parties[1], "B", SecretInteger)
```

Matrix B is 2x4, owned by Party1, named "B", and also of type SecretInteger.

4. Computation Section

Perform the matrix multiplication using the @ operator:

```python
    result = A @ B
```

This computes the matrix product of A and B.

5. Output Section

Produce the output for Party2 with the variable name "myoutput" using na.output:

python
```
    return result.output(parties[2], "myoutput")
```

This line specifies that Party2 will receive the output named "myoutput".

With this structure in place, you can build and test your program using:
bash
```
nada build
```

Using Nada Numpy with Nillion Network

Once your program is written, you can integrate it with the Nillion Network using the Python Nada Numpy client. The process is straightforward, similar to other examples. For detailed instructions and the complete code, refer to the GitHub repository.

First, import the necessary modules:
python
```
import numpy as np
import nadanumpy.client as naclient
```

Then, use naclient.array to format your arrays for upload to the Nillion Network:
python
```
A = np.ones((3, 2))  Sample numpy array with ones of shape 3x2
storedsecret = nillion.Secrets(naclient.array(A, "A"))
```

That’s it! You’ve successfully created, built, and integrated a Nada Numpy program with the Nillion Network, focusing on matrix multiplication with the @ operator.

For more examples, please visit our Github Repository Examples.

# nada-numpy-operators.md:

Nada Array Operators

Arithmetic Operators

Operator	Example	Operation
+	array1 + array2	Element-wise addition with broadcasting.
-	array1 - array2	Element-wise subtraction with broadcasting.
	array1 array2	Element-wise multiplication with broadcasting.
/	array1 / array2	Element-wise division with broadcasting.
@	array1 @ array2	Matrix multiplication with another NadaArray.
+=	array1 += array2	Inplace element-wise addition with broadcasting.
-=	array1 -= array2	Inplace element-wise subtraction with broadcasting.
=	array1 = array2	Inplace element-wise multiplication with broadcasting.
/=	array1 /= array2	Inplace element-wise division with broadcasting.
@=	array1 @= array2	Inplace matrix

multiplication with another NadaArray. |

Logical Operators

Operator	Example	Operation
==	array1 == array2	Element-wise equality comparison.
<	array1 < array2	Element-wise less than comparison.
<=	array1 <= array2	Element-wise less than or equal to comparison.
>	array1 > array2	Element-wise greater than comparison.
>=	array1 >= array2	Element-wise greater than or equal to comparison.
-array	-array1	Element-wise negation.
array n	array1 n	Element-wise exponentiation.

Other Operators

Operator	Example	Operation
compress	array.compress(condition)	Compress elements based on condition.
copy	array.copy()	Create a copy of the array.
cumprod	array.cumprod()	Cumulative product of elements.
cumsum	array.cumsum()	Cumulative sum of elements.
diagonal	array.diagonal(offset=1)	Retrieve diagonal elements.
fill	array.fill(value)	Fill array with a scalar value.
flatten	array.flatten()	Flatten array into a 1D array.
item	array.item(index)	Get item at specified index.
itemset	array.itemset(index, value)	Set item at specified index to value.
prod	array.prod()	Product of all elements.
put	array.put(indices, values)	Put values into specified indices.
ravel	array.ravel()	Flatten array into a 1D array.
reshape	array.reshape(shape)	Reshape array to specified shape.
resize	array.resize(newshape)	Resize array to new shape.
squeeze	array.squeeze()	Remove single-dimensional entries from shape.
sum	array.sum()	Sum of all elements.
swapaxes	array.swapaxes(axis1, axis2)	Swap two axes of the array.
take	array.take(indices)	Take elements from array at specified indices.
tolist	array.tolist()	Convert array to a

Python list. |

trace	array.trace(offset=0)	Compute the trace of the array.
transpose	array.transpose()	Transpose array dimensions.
base	array.base	Base object of the array.
data	array.data	Data pointer to the array's memory.
flags	array.flags	Information about the memory layout of the array.
flat	array.flat	1D iterator over the array.
itemsize	array.itemsize	Size of a single element in bytes.
nbytes	array.nbytes	Total bytes consumed by the array's elements.
ndim	array.ndim	Number of array dimensions.
shape	array.shape	Shape of the array.
size	array.size	Number of elements in the array.
strides	array.strides	Strides of the array.
T	array.T	Transposed view of the array.

For more examples, please visit our Github Repository Examples.

# nada-numpy-rationals.md:

Rational Numbers Tutorial

This tutorial shows how to use Nada Numpy Rational datatypes to work with fixed-point numbers in Nada.

Notions

This tutorial uses fixed-point numbers as it is the only available way to use Fixed-Point numbers in Nada. The representation of a fixed-point number uses integer places to represent decimal digits. Thus, every number is multiplied by a scaling factor, that we refer to as SCALE, generally chosen to be a power of 2 where the exponent represents the number of bits. In a nutshell, SCALE is going to reserve a number of bits for the exponent.

If we want to input a variable a = float(3.2), we need to first encode it. For that, we will define a new variable a' which is going to be the scaled version. In this case, the scaling factor (to simplify) is going to be 3 bits so our scale would be SCALE = 2 3 8. By multiplying our variable a with SCALE we obtain the encoded value. To decode, we just need to make the division by the scale.

```python
BITS = 3
SCALE = 2  BITS2 #8
a = float(3.2)

Encoding
aencoded = round(a  SCALE) round(3.2  8) = 26

Decoding
```

adecoded == aencoded / SCALE 26 / 8 = 3.25

Thus, to introduce a value with 3 bits of precision, we would be inputting 26 instead of 3.2. Note that, the larger the BITS precission, the better result we would obtain when decoding.

:::tip

Nada Numpy uses a default value of 16 bits for decimal scale. If you want to change it, you can do so with:

python
```
na.setlogscale(BITS)
```

:::

Example

python
```
from nadadsl import

import nadanumpy as na


def nadamain():
    We define the number of parties
    parties = na.parties(3)

    We use na.SecretRational to create a secret rational number for party 0
    a = na.secretrational("myinput0", parties[0])

    We use na.SecretRational to create a secret rational number for party 1
    b = na.secretrational("myinput1", parties[1])

    This is a compile time rational number
    c = na.rational(1.2)

    The formula below does operations on rational numbers and returns a rational
number
    It's easy to see that (a + b - c) is both on numerator and denominator, so
the end result is b
    out0 = ((a + b - c)  b) / (a + b - c)

    return [
        Output(out0.value, "myoutput0", parties[2]),
    ]
```

Let’s break this down step-by-step.

1. Import Section

Start by importing the necessary modules. We need Nada DSL and Nada Numpy, so we import them like this:

python
```
from nadadsl import
import nadanumpy as na
```

Next, define the main function for our program:

```
python
def nadamain():
```

2. Party Declaration Section

In this section, we’ll declare the parties involved in our computation. Parties represent the different participants in our Nada program. Nada Numpy makes it easy to create multiple parties with the na.parties function:

```
python
    parties = na.parties(3)
```

This line creates a list of three parties named: Party0, Party1, and Party2.

3. Input Declaration Section

Now, let’s define our inputs. We’ll create two secret rational numbers using na.SecretRational. This function allows us to define rational numbers with a specific owner and name.

```
python
    a = na.secretrational("myinput0", parties[0])
```

This line creates a secret rational number a owned by Party0, named “myinput0”.

Similarly, we define the second secret rational number b:

```
python
    b = na.secretrational("myinput1", parties[1])
```

This secret rational number is owned by Party1, named “myinput1”.

We also define a compile-time rational number c:

```
python
    c = na.rational(1.2)
```

This line creates a compile-time rational number c with a value of 1.2.

4. Computation Section

With our inputs ready, we can perform computations. In this example, we compute a specific rational operation.

```
python
    out0 = ((a + b - c)  b) / (a + b - c)
```

This line performs the operation (a + b - c) b / (a + b - c), which simplifies to b because the numerator and denominator are the same.

5. Output Section

Finally, we need to produce the output. Since SecretRational is not a base Nada type, we use the Output method to generate the output. This method takes the value, output party, and output variable name as arguments:

```
python
    return [
        Output(out0.value, "myoutput0", parties[2]),
    ]
```

In this case, we specify that the output party will be Party2 and the name of the output variable will be "myoutput0".

With everything in place, we can build and test our program:

```bash
nada build
```

(Optional) Next, ensure that the program functions correctly by testing it with:

```bash
nada test
```

Finally, we can call our Nada program via the Nillion Python client by running:

```bash
python3 main.py
```

And that’s it! You’ve successfully created, built, and integrated a Nada Numpy Rational numbers program.

For more examples, please visit our Github Repository Examples.

# nada-run.md:

nada-run

The nada-run tool executes a Nada program using the Nada VM with the cryptographic algorithms but without the peer-to-peer (p2p) network.

```bash
Usage: nada-run [OPTIONS] --prime-size <PRIMESIZE> <PROGRAMPATH>

Arguments:
  <PROGRAMPATH>
          Program path to the compiled program file

Options:
  -p, --prime-size <PRIMESIZE>
          Prime size in bits; 256 is a good default

  -n, --network-size <NETWORKSIZE>
          The size of the local network

          [default: 3]

  -d, --polynomial-degree <POLYNOMIALDEGREE>
          The degree of the polynomial used

          [default: 1]

      --int-secret <INTEGERSECRETS>
          An integer secret.

          These must follow the pattern <name>=<value>.

          [aliases: i]

      --uint-secret <UNSIGNEDINTEGERSECRETS>
```

```
          An unsigned integer secret.

          These must follow the pattern <name>=<value>.

          [aliases: ui]

      --secrets-path <SECRETSPATH>
          A path to load secrets from

      --int-public-variable <INTEGERPUBLICVARIABLES>
          An integer public variable.

          These must follow the pattern <name>=<value>.

          [aliases: ip]

      --uint-public-variable <UNSIGNEDINTEGERPUBLICVARIABLES>
          An unsigned integer public variable.

          These must follow the pattern <name>=<value>.

          [aliases: uip]

      --public-variables-path <PUBLICVARIABLESPATH>
          A path to load secrets from

  -h, --help
          Print help (see a summary with '-h')

  -V, --version
          Print version
```

# nada-test.md:

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';

nada-test: Nada Testing Framework

nada-test is a powerful and flexible testing framework for Nada programs. It enables developers to write dynamic tests that are generated at runtime, offering more flexibility than traditional YAML test files.

Features

- Dynamic test generation at runtime
- Easy-to-write test functions using Python
- Seamless integration with nada projects
- Support for both standalone usage and integration with pytest
- Flexible input and output specification for test cases

Installation

You can install nada-test in a Nada project using pip:

```
bash
pip install nada-test
```

Setup

To use nada-test as your test framework, you’ll need to configure it in your nada-project.toml file. This allows the nada test command to automatically run your tests using the nada-test framework. Here’s how you can set it up:

<Tabs>
<TabItem value="basic" label="Basic configuration" default>
Add the following to your nada-project.toml file to set nada-test as your default test runner and point it to the ./tests directory:

```
toml
[testframework.nada-test]
command = "nada-test ./tests"
```

This setup ensures that all tests inside the ./tests directory will be executed when you run nada test.
</TabItem>

<TabItem value="custom" label="Custom test directories">
If you have tests in multiple directories, you can specify them in the configuration as well:

```
toml
[testframework.nada-test]
command = "nada-test ./customtests ./moretests"
```

This allows you to organize your tests across different directories, and nada test will run all tests from the specified paths.
</TabItem>
</Tabs>

Writing tests

You can use nada-test to write both functional and class-based tests for your programs. Tests are decorated with @nadatest(program="programname") to specify which program is being tested. Below are two examples showing how to test a basic addition program.

Tests should be written in a Python file located in the directory you specified during the setup (e.g., ./tests or any custom test directory). Below are examples of how to test a basic addition program using both styles.

Functional style test

<Tabs>
<TabItem value="test" label="nada-test file" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/additiontest.py#L4-L11

</TabItem>

<TabItem value="program" label="Nada program">
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/addition.py

</TabItem>
</Tabs>

Class-based test

<Tabs>
<TabItem value="test" label="nada-test file" default>

python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/additiontest.py#L13-L21

</TabItem>

<TabItem value="program" label="Nada program">
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/addition.py

</TabItem>
</Tabs>

Running tests

After writing your tests in the specified directories, you can run them using the following command:

```
nada test
```

This will execute all the tests configured in your nada-project.toml file. The output will show you the results of your test suite.

# nada.md:

import VenvSetup from './\nada-venv-setup.mdx';

nada

nada is the tool to manage Nada projects. It can:

- Create a new project,
- Compile Nada programs,
- Get program-requirements, the runtime requirements of the preprocessing elements of a program,
- Run Nada programs (also in debug mode),
- Test a program (also in debug mode),
- Generate a test from a program.

Requirements

- Install the Nillion SDK to access the nada tool

Usage

Create a new project

You can create a new nada project by running

```
nada init <project-name>
```

This will create a new directory with the project name and the following structure:

```
nada-project.toml  File to configure the project
src/               Directory to store the programs
src/main.py        nada program
target/            Directory where nada will produce the compiled programs
```

```
tests/             Directory to store the tests
```

nada-project.toml file

This file defines the nada project, it looks like this

```
toml
name = "<project name>" name of the project
version = "<version>"       version of the project
authors = [""]          authors of the project

[[programs]]            program entry, you can have several of these
path = "src/main.py"    path to the program
primesize = 128        prime size to use in the execution of the program
```

nil-sdk.toml file

We recommend creating a nil-sdk.toml file in the project directory to specify the version of the Nada SDK to use.
The file should have the following format:

```
toml
version = "<version>"
```

This will ensure that when you call nada or any other Nillion SDK command, it will use the version specified in that file.

Set up a virtual environment

<VenvSetup/>

1. Create a python virtual environment

```
python3 -m venv .venv
```

2. Activate the python virtual environment

```
source .venv/bin/activate
```

3. Install nada-dsl

```
pip install --upgrade nada-dsl
```

Add a new Nada program to the project

To add a new program to the project, you can create a new file in the src directory and add a new entry in the nada-project.toml file.
There is the option to use another name for your program using the name field in the program entry.

Let's say that the new file is called newprogram.py and you want to call it myprogram, then you can add a new entry in the nada-project.toml file like this:

```
toml
[[programs]]
```

```
path = "src/newprogram.py"
name = "myprogram"
primesize = 128
```

Build (compile) a program

To build all the programs in the project, you can run

```
nada build
```

Note: this will only build the programs specified in the nada-project.toml file.
Visit the section Add a new program to the project for more information.

Also you can specify which program to build by passing the program name as an argument

```
nada build <program-name>
```

The build command will produce a <program name>.nada.bin file in the target directory.
You can use this file to upload and run it on the Nillion Network.

Get program requirements

The program-requirements command prints the runtime requirements of the preprocessing elements of a specific program.

```
nada program-requirements <program-name>
```

Example program and program-requirements:

```
from nadadsl import

def nadamain():
    party1 = Party(name="Party1")
    party2 = Party(name="Party2")
    party3 = Party(name="Party3")
    a = SecretInteger(Input(name="A", party=party1))
    b = SecretInteger(Input(name="B", party=party2))

    result = a + b

    return [Output(result, "myoutput", party3)]
```

Here are the program requirements of the 3 Party addition program above:

```
Requirements:
 ProgramRequirements {
    alphaelements: 2,
    runtimeelements: {
        Lambda: 2,
    },
}
```

Generate a test file

To be able to run or test programs we need a test file with the inputs and expected outputs.
nada has the ability to generate a test file with example values for you.
To generate a test file you can run

```
nada generate-test --test-name <test-name> <program-name>
```

It will generate a test file with the name <test-name>.yaml in the tests directory.

You should edit the test file to change the inputs and the expected outputs.

Run a program

To run a program using nada you need a test file, visit the Generate a test file section to see how to generate it.

Run in normal mode

To execute a program you can run

```
nada run <test-name>
```

It will run the program associated with that test file and print the output.
Similarly to the nada-run tool, it executes a Nada program using the Nada VM with the cryptographic algorithms but without the peer-to-peer (p2p) network.

Run in debug mode

nada has the ability to run a program in debug mode.
In this mode, all operations are computed in plain text, so you can inspect the intermediary values.
When running in debug mode, you can view the operations performed in the program along with their corresponding values, as it does not execute the cryptographic algorithms.

To execute a program in debug mode you can run

```
nada run --debug <test-name>
```

This will run the program associated with that test file and print all operations, values and the output.

Run and get protocols model text file

To execute a program and have a protocols model text file <test-name>.nada.protocols.txt generated for your program, run with -protocols-text or -p

```
nada run <test-name> -protocols-text
```

Example program and generated protocols model text file:

```
from nadadsl import

def nadamain():
    party1 = Party(name="Party1")
    party2 = Party(name="Party2")
    party3 = Party(name="Party3")
    a = SecretInteger(Input(name="A", party=party1))
    b = SecretInteger(Input(name="B", party=party2))

    result = a + b

    return [Output(result, "myoutput", party3)]
```

Here is the generated protocols model text file for the 3 Party addition program above:

```
Inputs:

iaddr(0): A (ref 0) (sizeof: 1)
iaddr(1): B (ref 0) (sizeof: 1)


Literals:


Protocols:
rty(ShamirShareInteger) = P2S iaddr(0)
result = a + b  -> main.py:10
rty(ShamirShareInteger) = P2S iaddr(1)
result = a + b  -> main.py:10
rty(ShamirShareInteger) = ADD addr(2) addr(3)
result = a + b  -> main.py:10


Outputs:
addr(4): myoutput
```

Run and get bytecode in text format

To execute a program and have a bytecode text file <test-name>.nada.bytecode.txt generated for your program, run with -bytecode-text or -b

```
nada run <test-name> -bytecode-text
```

Here is the generated bytecode text file for the 3 Party addition program above:

```
Header:
Party: Party1 id(0)   party1 = Party(name="Party1")  -> main.py:4
 Inputs:
  iaddr(0) rty(SecretInteger) = Input(A)
a = SecretInteger(Input(name="A", party=party1))  -> main.py:7
 Outputs:

Party: Party2 id(1)   party2 = Party(name="Party2")  -> main.py:5
 Inputs:
  iaddr(1) rty(SecretInteger) = Input(B)
b = SecretInteger(Input(name="B", party=party2))  -> main.py:8
```

```
 Outputs:

Party: Party3 id(2)   party3 = Party(name="Party3")  -> main.py:6
 Inputs:
 Outputs:
  oaddr(0) rty(SecretInteger) = Output(myoutput) addr(2)
-> main.py:12

Literals:



Operations:
addr(0) rty(SecretInteger) = Load iaddr(0)
a = SecretInteger(Input(name="A", party=party1))  -> main.py:7
addr(1) rty(SecretInteger) = Load iaddr(1)
b = SecretInteger(Input(name="B", party=party2))  -> main.py:8
addr(2) rty(SecretInteger) = Addition addr(0) addr(1)
```

Test a program

To test a program using nada you need a test file, visit the Generate a test file section to see how to generate it.

Test in normal mode

To run all the tests in the project you can run

```
nada test
```

To run a single test you can run

```
nada test <test-name>
```

This will run the program associated with that test file and check if the output matches what's expected.

Test in debug mode

You can also run a test in debug mode, to know more about debug mode visit the Run in debug mode section.
To run a test in debug mode you can run

```
nada test --debug <test-name>
```

# network-configuration.md:

```
import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import NetworkTable from '@site/src/components/Networks/index';
import {PythonTestnetEnv, ReactTestnetEnv} from
'@site/src/components/Networks/TestnetEnv';
```

Network Configuration

Testnet

Use the Testnet configuration to connect to the integrated Nillion Testnet.
Check out Testnet wallet and faucet guides here.

<Tabs>

<TabItem value="table" label="Network Table" default>
<NetworkTable/>
</TabItem>

<TabItem value="python" label="Python .env">
<PythonTestnetEnv/>
</TabItem>

<TabItem value="react" label="React .env">
<ReactTestnetEnv/>
</TabItem>

</Tabs>

Local Devnet

Use the nillion-devnet SDK tool to spin up a devnet that you can interact with locally while you keep the process running:

```
nillion-devnet --seed my-seed
```

You will see an output like this:

```
nillion-devnet --seed my-seed
ℹ️ cluster id is 222257f5-f3ce-4b80-bdbc-0a51f6050996
ℹ️ using 256 bit prime
ℹ️ storing state in /var/folders/1/2yw8krkx5q5dn2jbhx69s4r0000gn/T/.tmpU00Jbm
(62.14Gbs available)
🏃 starting nilchain node in:
/var/folders/1/2yw8krkx5q5dn2jbhx69s4r0000gn/T/.tmpU00Jbm/nillion-chain
⛓  nilchain JSON RPC available at http://127.0.0.1:48102
⛓  nilchain gRPC available at localhost:26649
🏃 starting node 12D3KooWMGxv3uv4QrGFF7bbzxmTJThbtiZkHXAgo3nVrMutz6QN
⏳ waiting until bootnode is up...
🏃 starting node 12D3KooWKkbCcG2ujvJhHe5AiXznS9iFmzzy1jRgUTJEhk4vjF7q
🏃 starting node 12D3KooWMgLTrRAtP9HcUYTtsZNf27z5uKt3xJKXsSS2ohhPGnAm
💰 funding nilchain keys
📝 nillion CLI configuration written to /Users/steph/Library/Application
Support/nillion.nillion/config.yaml
🌄 environment file written to /Users/steph/Library/Application
Support/nillion.nillion/nillion-devnet.env
```

Copy the path printed after "🌄 environment file written to" and open the file to see your local devnet network configuration

# network.md:

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import SupportedWallets from './\testnetsupportedwallets.mdx';
import BlockExplorers from './\testnetblockexplorers.mdx';

Nillion Network

The Nillion Network allows developers to build applications and services that leverage privacy-enhancing technologies (PETs) such as secure multi-party computation (MPC) to perform blind computations over private user data (without requiring that data to be revealed to the network or to other users).

Dual Network Architecture

!Nillion's dual network architecture diagram

The Nillion Network consists of two parallel, interdependent networks: a Coordination Layer and an Orchestration Layer.

 The NilChain network coordinates payments for storage operations and blind computations performed on the network.
 The Petnet harnesses PETs such as MPC to protect data at rest and to enable blind computations that can operate on that data.

The Nillion Network operates without a global shared state or consensus on order. Instead, its design is oriented around supporting storage of private high-value data and computation over that data while decentralizing trust among the nodes in the network.

Live Testnet

Consistent with the dual network architecture described above, the Nillion Network testnet (or simply Nillion testnet) consists of two interdependent testnet instances: the NilChain testnet and the Petnet testnet.

NilChain Testnet

The NilChain testnet is live! Follow our Testnet Guides to create a wallet connected to the Nillion testnet, use the testnet faucet, and send NIL testnet tokens.

```
    <Tabs>
        <TabItem value="wallets" label="Supported wallets" default>

            <SupportedWallets/>

            Wallet Guide: Creating a Nillion Wallet

        </TabItem>
        <TabItem value="explorers" label="Block explorers">

            <BlockExplorers/>

            Transaction Guide: Sending NIL and Using a Block Explorer

        </TabItem>

        <TabItem value="chaininfo" label="chain-info.json" default>

        NilChain testnet chain info:

        json reference showGithubLink
        https://github.com/chainapsis/keplr-chain-registry/blob/main/cosmos/
nillion-chain-testnet.json



        </TabItem>
    </Tabs>
```

Petnet Testnet

The Petnet testnet is also live! Builders can now build their blind applications and then connect them to the Petnet testnet to make them accessible to the whole community.

Local Devnet

Builders that prefer to work locally can use the Nillion SDK to connect their blind applications to a local instance of the nillion-devnet. Builders have the option to connect these applications to the Petnet testnet once they are ready.

# nillion-client.md:

import DocCardList from '@theme/DocCardList';

Nillion Client

There are 3 Nillion Clients - a Python, JavaScript, and CLI Client. Each Nillion Client provides APIs that you can use for generating user and node keys, and managing programs, secrets, and permissions on the Nillion Network.

<DocCardList/>

NillionClient

Create an instance of NillionClient

NillionClient creates an instance of the Nillion Client library with node key, bootnodes, connection mode, user key, and payments parameters. This client instance interacts with programs, secrets, and permissions on the network as the user via their user key.

Programs

Store a program

storeprogram uploads a compiled Nada program to the Nillion Network.&#x20;

storeprogram returns the stored program's programid from the network

Run a stored program

compute runs a stored Nada program by programid against stored secrets by storeid and secretname and/or secrets provided by the user when running compute. A user needs permission to compute on stored secrets for a specific program.

compute returns the program result from the network

Program permissions

<table><thead><tr><th width="190">Operation</th><th width="235">Permission needed</th><th>Description</th></tr></thead><tbody><tr><td><code>storeprogram</code></td><td>-</td><td>Any user with an allowlisted peer id can store a program on the Nillion Network</td></tr><tr><td><code>compute</code></td><td>addcomputepermissions</td><td>Allows a user to compute on the secret for a specific program</td></tr></tbody></table>

Secrets

Store secrets

storesecrets uploads permissioned secrets to the Nillion Network. Each secret value is uploaded with a secretname that is set by the user. Every node in the network stores a particle of these secrets.

storesecrets returns the secret's storeid from the network

Set permissions while storing secrets

The user storing the secrets can give "default permissions" of the secrets with defaultforuser(userid). Any userid with these "default permissions" will have permission to retrieve and update secret permissions.

The user storing the secret can give other userids limited permissions to the secrets by specifying the userid to allowlist and the intended secret permissions (retrieve / update / delete / compute) to grant that user.

<table><thead><tr><th width="242">Permission</th><th>Operation unlocked</th><th>Description</th></tr></thead><tbody><tr><td>defaultforuser</td><td><code>permissions</code></td><td>Allows a user to update permissions for a secret</td></tr><tr><td>addretrievepermissions</td><td><code>retrievesecret</code></td><td>Allows a user to read or retrieve a secret</td></tr><tr><td>addupdatepermissions</td><td><code>updatesecret</code></td><td>Allows a user to update a secret</td></tr><tr><td>adddeletepermissions</td><td><code>deletesecret</code></td><td>Allows a user to delete a secret</td></tr><tr><td>addcomputepermissions</td><td><code>compute</code></td><td>Allows a user to compute on a secret as an input to a specific program id</td></tr></tbody></table>

Retrieve a secret

retrievesecret retrieves a secret by storeid and secretname. A user needs to have secret retrieve permissions (addretrievepermissions) to retrieve a stored secret.

retrievesecret returns the secret from the network

Update a secret

updatesecret updates a secret value by storeid. A user needs to have secret update permissions (addupdatepermissions) to update a stored secret.

Delete a secret

deletesecret deletes a secret value by storeid. A user needs to have secret delete permissions (adddeletepermissions) to delete a stored secret.

Permissions

Update permissions

updatepermissions replaces the permissions of a secret by storeid. A user needs to have defaultforuser permissions (grants the ability to retrieve and update secret permissions) to update permissions of a stored secret.&#x20;

The user updating permissions for the secrets can give other userids limited permissions to the secrets by specifying the userid to allowlist and the intended secret permissions to grant that user.

# nillion-devnet.md:

nillion-devnet

The nillion-devnet tool creates a Nillion network devnet that you can interact with while you keep the process running.

Run Devnet

```bash
Usage: nillion-devnet [OPTIONS]

Options:
  -n, --node-count <NODECOUNT>
          The number of nodes in the devnet

          [default: 3]

  -c, --cluster-id <CLUSTERID>
          The uuid of the cluster.

          A random uuid will be used if none is provided, honoring the --seed
parameter.

  -d, --state-directory <STATEDIRECTORY>
          The directory where the node's states is stored.

          A temporary directory will be used if none is provided.

  -s, --seed <SEED>
          The seed to use, if any, for keys and cluster ids.

          If none is provided, node keys and the cluster id will be randomized.

  -p, --prime-bits <PRIMEBITS>
          The number of bits in the prime number to be used

          [default: 256]

  -b, --bind-address <BINDADDRESS>
          The address to bind to

          [default: 127.0.0.1]

  -h, --help
          Print help (see a summary with '-h')

  -V, --version
          Print version
```

Devnet outputs

Running a local devnet outputs values you can use to run nillion against your local devnet rather than the network

- devnet id
- blockchain node endpoint
- node ids
- wallet keys: 14 private keys written to a file
- payment configuration (blockchain info) written to a file
  - blockchainrpcendpoint
  - chainid
  - paymentsscaddress
  - blindingfactorsmanagerscaddress
- bootnode
- websocket

Spin down local devnet

To stop the local devnet, run

```bash
killall nillion-devnet
```

# nillion-sdk-and-tools.md:

import SdkInstallation from './\sdk-installation.mdx';

Nillion SDK and Tools

The Nillion SDK includes CLI tools for generating user and node keys, compiling Nada programs, running programs locally, running a nillion devnet, and connecting to the Nillion network.

Installation

<SdkInstallation/>

Nillion SDK tools

After installation, the following SDK tools are available globally:

- nilup: a tool to install the Nillion SDK and manage Nillion SDK versions.
- nillion: a cli-based Nillion Client and tool for interacting with the Nillion Network from the command line to generate user keys, generate node keys, store secrets, retrieve secrets, store programs, compute on secrets, and fetch information about clusters and nodes.
- nillion-devnet: a tool that allows you to spin up and interact with a local test Nillion network that is completely isolated within your computer
- node-key2peerid: a tool that creates a peer id from your node key
- nada: a tool to manage Nada projects (create project, compile, run, and test programs, generate tests, etc.).
- nada-run: a tool that executes programs against a stripped down version of a Nillion devnet
- pynadac: a tool that compiles Nada programs; pynadac takes an input program defined in Nada and produces a compiled version of it ready to be run with nada-run or stored on the Nillion Network

Command structure

Nillion SDK tool commands follow a structured format:

```bash

<tool> [options] <command>
```

For example, to generate a user key using the nillion command, run:

```bash
nillion user-key-gen user.key
```

To get full usage details including a comprehensive list of global commands and options available for a specific tool, run:

```bash
<tool> --help
```

For example, to view the available commands for the nada tool, run:

```
    bash
    nada --help
```

# nillion.md:

nillion

The nillion tool is a command line version of the Nillion Client. We refer to the tool as the CLI Client.

The nillion tool can be used to generate user keys, generate node keys, store secrets, retrieve a secret, store a program, compute on a program, get cluster information, check the preprocessing pool status, and display node and user ids from keys.

nillion can be run against the Nillion Network or against a local cluster spun up with nillion-devnet. If nillion is running against your local cluster, use the cluster id, bootnodes, smart contract addresses, chain id, rpc url endpoint values that are local cluster outputs as nillion options.

Installation

Follow instructions to install the Nillion SDK and Tools, which include nillion tool installation

Usage

```
bash
Usage: nillion [OPTIONS] -b <BOOTNODES> <COMMAND>
Commands:
  store-values               Store values in the network
  retrieve-value             Retrieve a value from the network
  store-program              Store a program in the network
  compute                    Perform a computation in the network
  cluster-information        Fetch the cluster's information
  delete-values              Delete values from the network
  preprocessing-pool-status  Fetch the preprocessing pool status for a cluster
  inspect-ids                Display the node/user ids derived from the provided
keys
  shell-completions          Generate shell completions
  node-key-gen               Generate Node keys
  user-key-gen               Generate User keys
  retrieve-permissions       Retrieve permissions for stored secrets
  set-permissions            Set permissions on a stored secrets
  help                       Print this message or the help of the given
subcommand(s)

Options:
      --user-key <USERKEY>
          The user key in base58

      --user-key-seed <USERKEYSEED>
          The seed to be used to derive user key

  -u, --user-key-path <USERKEYPATH>
          The path to the file that contains the user key in base58

      --node-key <NODEKEY>
```

```
          The node key in base58

      --node-key-seed <NODEKEYSEED>
          The seed to be used to derive node key

  -n, --node-key-path <NODEKEYPATH>
          The path to the file that contains the node key in base58

  -b <BOOTNODES>
          A list of bootnodes to connect to.

          This needs to use libp2p multiaddress format.

  -w <WEBSOCKETBOOTNODES>
          A list of websocket bootnode addresses to connect to.

          This needs to use libp2p multiaddress format.

  -l, --listen-address <LISTENADDRESS>
          The address to listen on.

          If none is provided, the client will be expected to dial all of its
peers (not receive incoming connections).

      --nilchain-rpc-endpoint <NILCHAINRPCENDPOINT>
          The nilchain RPC endpoint

      --nilchain-private-key <NILCHAINPRIVATEKEY>
          The nilchain payments private key

      --gas-price <GASPRICE>
          The gas price to use, in unil units

  -h, --help
          Print help (see a summary with '-h')

  -V, --version
          Print version
```

Generate user key with nillion

Generate a Nillion user key and store it in a file

user-key-gen usage

```bash
nillion user-key-gen <FILENAME>
```

```
Arguments:
  <FILENAME>  key filename

Options:
  -s, --seed <SEED>  seed to generate the key
  -h, --help         Print help
```

Generate node key with nillion

Generate a Nillion node key and store it in a file

node-key-gen usage

```bash
nillion node-key-gen <FILENAME>
```

```
Arguments:
  <FILENAME>  key filename

Options:
  -s, --seed <SEED>  seed to generate the key
  -h, --help         Print help
```

Store values with nillion

Store values in the network

store-values usage

```bash
nillion \
    -b <YOURBOOTNODE> \
    --nilchain-private-key <YOURPRIVATEKEY> \
    --nilchain-rpc-endpoint <YOURRPCENDPOINT> \
    --node-key-seed <YOURNODEKEYSEED> \
    --user-key-seed <YOURUSERKEYSEED> \
    store-values \
    --secret-integer <name>=<value> \
    --cluster-id <YOURCLUSTERID> \
    --ttl-days <NUMDAYS>
```

```bash
Arguments:
  [PROGRAMID]
          The program id that the store is for, if any

Options:
  -i, --public-integer <INTEGERS>
          An integer public variable.

          These must follow the pattern <name>=<value>.

      --public-unsigned-integer <UNSIGNEDINTEGERS>
          An unsigned integer public variable.

          These must follow the pattern <name>=<value>.

          [aliases: ui]

      --secret-integer <SECRETINTEGERS>
          An integer secret.

          These must follow the pattern <name>=<value>.

          [aliases: si]

      --secret-unsigned-integer <SECRETUNSIGNEDINTEGERS>
          An unsigned integer secret.

          These must follow the pattern <name>=<value>.

          [aliases: sui]
      --array-public-integer <ARRAYINTEGERS>
          An array of integer public variables

          The expected pattern is <name>=<comma-separated-value>.

          Example: array1=1,2,3

          [aliases: ai]

      --array-public-unsigned-integer <ARRAYUNSIGNEDINTEGERS>
          An array of unsigned integer public variables

          The expected pattern is <name>=<comma-separated-value>.

          Example: array1=1,2,3

          [aliases: aui]

      --array-secret-integer <ARRAYSECRETINTEGERS>
          An array of integer secrets

          The expected pattern is <name>=<comma-separated-value>.

          Example: array1=1,2,3

          [aliases: asi]

      --array-secret-unsigned-integer <ARRAYSECRETUNSIGNEDINTEGERS>
          An array of unsigned integer secrets

          The expected pattern is <name>=<comma-separated-value>.

          Example: array1=1,2,3

          [aliases: asui]

      --secret-blob <SECRETBLOBS>
          A blob secret.

          These must follow the pattern <name>=<value> and the value must be
encoded in base64.

          [aliases: sb]

      --nada-values-path <NADAVALUESPATH>
          A path to load secrets from

  -t, --ttl-days <TTLDAYS>
          The time to live for the values in days

  -c, --cluster-id <CLUSTERID>
          The cluster id to perform the operation on

      --authorize-user-execution <AUTHORIZEUSEREXECUTION>
          Give execution access to this user on the secret we're uploading

  -h, --help
          Print help (see a summary with '-h')
```

Retrieve a value with nillion

Retrieve a value from the network

retrieve-secret usage

```
bash
nillion \
    -b <YOURBOOTNODE> \
    --nilchain-private-key <YOURPRIVATEKEY> \
    --nilchain-rpc-endpoint <YOURRPCENDPOINT> \
    --node-key-seed <YOURNODEKEYSEED> \
    --user-key-seed <YOURUSERKEYSEED> \
    retrieve-value \
    --cluster-id <YOURCLUSTERID> \
    --secret-id <YOURSECRETNAME> \
    --store-id <YOURSTOREID>
```

```
bash
Options:
  -c, --cluster-id <CLUSTERID>  The cluster id to perform the operation on
      --store-id <STOREID>      The store id to retrieve
      --secret-id <SECRETID>    The specific secret id to be retrieved
  -h, --help                     Print help
```

Store a program with nillion

Name and store a compiled Nada program on the network.

store-program usage

```
bash
nillion \
    -b <YOURBOOTNODE> \
    --nilchain-private-key <YOURPRIVATEKEY> \
    --nilchain-rpc-endpoint <YOURRPCENDPOINT> \
    --node-key-seed <YOURNODEKEYSEED> \
    --user-key-seed <YOURUSERKEYSEED> \
    store-program \
    --cluster-id <CLUSTERID> \
    <PROGRAMPATH> <PROGRAMNAME>
```

```
bash
Arguments:
  <PROGRAMPATH>
  <PROGRAMNAME>  The name of the program
```

Compute on a program with nillion

Compute on a stored program by program id with stored secrets, secrets and public variables from files, or secrets and public variables input directly in the command

compute usage

```
bash
nillion \
    -b <YOURBOOTNODE> \
    --nilchain-private-key <YOURPRIVATEKEY> \
    --nilchain-rpc-endpoint <YOURRPCENDPOINT> \
    --node-key-seed <YOURNODEKEYSEED> \
    --user-key-seed <YOURUSERKEYSEED> \
  compute \
  [OPTIONS] \
  --cluster-id <CLUSTERID> \
  <PROGRAMID>


bash
Arguments:
  <PROGRAMID>
          The id of the program to be run

Options:
  -c, --cluster-id <CLUSTERID>
          The cluster id to perform the operation on

      --store-id <STOREIDS>
          A store secret id to be used..

      --result-node <RESULTNODES>
          A result node's party id.

          Not providing any result nodes means the dealer node will be the
defaulted as the result node.

      --result-node-name <RESULTNODENAMES>
          A result node's name

      --bind-dealer <BINDDEALER>


      --int-secret <INTEGERSECRETS>
          An integer secret.

          These must follow the pattern <name>=<value>.

          [aliases: i]

      --uint-secret <UNSIGNEDINTEGERSECRETS>
          An unsigned integer secret.

          These must follow the pattern <name>=<value>.

          [aliases: ui]

      --secrets-path <SECRETSPATH>
          A path to load secrets from

      --int-public-variable <INTEGERPUBLICVARIABLES>
          An integer public variable.

          These must follow the pattern <name>=<value>.

          [aliases: ip]

      --uint-public-variable <UNSIGNEDINTEGERPUBLICVARIABLES>
          An unsigned integer public variable.

          These must follow the pattern <name>=<value>.

          [aliases: uip]

      --public-variables-path <PUBLICVARIABLESPATH>
          A path to load secrets from

  -h, --help
```

```
          Print help (see a summary with '-h')
```

Get cluster information with nillion

Get info about the cluster

cluster-information usage

```
bash
nillion \
    -b <YOURBOOTNODE> \
    --nilchain-private-key <YOURPRIVATEKEY> \
    --nilchain-rpc-endpoint <YOURRPCENDPOINT> \
    --node-key-seed <YOURNODEKEYSEED> \
    --user-key-seed <YOURUSERKEYSEED> \
  cluster-information \
  --cluster-id <CLUSTERID> \

Arguments:
  <CLUSTERID>  The cluster id to query for

Options:
  -h, --help  Print help
```

Check the preprocessing pool status with nillion

Fetch the preprocessing pool status for a cluster

preprocessing-pool-status usage

```
bash
nillion \
    -b <YOURBOOTNODE> \
    --nilchain-private-key <YOURPRIVATEKEY> \
    --nilchain-rpc-endpoint <YOURRPCENDPOINT> \
    --node-key-seed <YOURNODEKEYSEED> \
    --user-key-seed <YOURUSERKEYSEED> \
  preprocessing-pool-status \
  --cluster-id <CLUSTERID> \

Arguments:
  <CLUSTERID>  The cluster id to query for

Options:
  -h, --help  Print help
```

Display node and user ids with nillion

Check the peer id of the node and the user id of the user.

inspect-ids usage

```
bash
nillion \
    -b <YOURBOOTNODE> \
    --nilchain-private-key <YOURPRIVATEKEY> \
    --nilchain-rpc-endpoint <YOURRPCENDPOINT> \
    --node-key-seed <YOURNODEKEYSEED> \
    --user-key-seed <YOURUSERKEYSEED> \
  inspect-ids
    --cluster-id <CLUSTERID> \
Options:
  -h, --help  Print help
```

# nillions-mpc-protocol.md:

Nillion’s MPC Protocol

Nillion’s novel MPC protocol extends the capabilities of Linear Secret-Sharing Schemes (LSSS) to perform some non-linear operations, specifically evaluating a sum of products of non-zero user inputs.

The protocol workflow is split into 2 phases:

Phase 1: Pre-processing to create shares

The pre-processing phase prepares for the network to securely handle the high value data so that future computations can be performed without revealing individual inputs.

The point of pre-processing is to generate and distribute shares (masks) for each factor and term in the sum of products expression using standard MPC techniques.

Pre-processing is independent of input values. This phase only depends on the number of inputs (factors and terms) so that the appropriate number of shares are created ahead of computation. &#x20;

Phase 2: Non-Interactive computation on masked factors

The computation phase is where the actual calculations on the private inputs are performed. Computation can be broken into 3 stages: input, evaluation, and output.

Input Stage

Shares generated during the pre-processing phase are distributed to parties. A party receives one share per input. Parties combine their inputs with shares to create masked factors. These multiplicatively homomorphic masked factors are broadcasted to the network, maintaining information-theoretic security (ITS).

Evaluation Stage

The evaluation stage utilizes the computationally homomorphic properties of the masked factors, enabling operations like addition and multiplication to be carried out directly on the masked data.&#x20;

Parties perform local calculations on masked factors.&#x20;

Output Stage

Parties reveal the result of local calculations. These results are aggregated to derive and output the final result of the multi-party computation.&#x20;

Protocol Features

Nillion’s novel MPC protocol achieves

- Non-linear arithmetic capabilities: The protocol can evaluate a sum of products of hidden inputs without leaking input information to any of the parties.
- Efficient pre-processing: The creation of shares is independent of input

values and only depends on the number of inputs (factors and terms) in each term.
- Asynchronous computation: The non-interactive nature of the computation phase aligns with asynchronous workflows and accelerates the process, as it does not require message exchanges between parties.
- ITS security: The protocol upholds the ITS inherent in LSSS, meaning it is secure against adversaries with unlimited computing resources and time.

Read the full technical paper on Nillion's MPC Protocol: Evaluation of Arithmetic Sum-of-Products Expressions in Linear Secret Sharing Schemes with a Non-Interactive Computation Phase

&#x20;

&#x20;

# nilup.md:

nilup

nilup is an installer and version manager for the Nillion SDK

To install nilup, run:

For the security-conscious, please download the install.sh script, so that you can inspect how it
works, before piping it to bash.

```
curl https://nilup.nilogy.xyz/install.sh | bash
```

If you install the Nillion SDK using nilup, all the Nillion SDK commands will be managed by nilup,
which will pick the version to use based on the next order, from highest to lowest priority:
the +<version> flag, the nil-sdk.toml file, and the global version set with the use command.

+<version> flag

When you call any Nillion SDK command like nada you can add a +<version> argument to use that version of nada.
For example if you run nada +0.0.4 init will run nada version 0.0.4.

nil-sdk.toml file

If there is a nil-sdk.toml or .nil-sdk.toml file in the current directory or parent directories, the Nillion SDK commands will use the version specified in that file.

The file should have the following format:

```
toml
version = "{version}"
```

Global version

If no version is specified in the command or in the nil-sdk.toml file, the Nillion SDK commands will use the global version set with the use command.
You can set the global version with the command

nilup use {version}

Usage

bash
```
Usage: nilup <COMMAND>

Commands:
  install            Install a version of the Nillion SDK
  uninstall          Uninstall a version of the Nillion SDK
  use                Set a global version of the Nillion SDK to be used
  list-available     List available versions of the Nillion SDK
  list-installed     List installed versions of the Nillion SDK
  instrumentation    Enable/Disable instrumentation
  shell-completions  Generate shell completions
  help               Print this message or the help of the given subcommand(s)

Options:
  -h, --help     Print help
  -V, --version  Print version
```

# node-key2peerid.md:

node-key2peerid

The node-key2peerid tool uses your node key or a node key file to tell you your node's peer id

Usage

bash
```
node-key2peerid [OPTIONS] [KEYFILE]
```

bash
```
Arguments:
  [KEYFILE]  node keypair file path to load

Options:
  -k, --key <KEY>    node keypair as base58 string
  -s, --seed <SEED>  seed to generate the keypair
  -h, --help         Print help
  -V, --version      Print version
```

# nucleus-builders-program.md:

---
description: Apply to the Nucleus Builders Program to start building on Nillion
---

Nucleus Builders Program

Nillion is on a mission to decentralize high value data across a wave of new web3 use cases and industries. We are working directly with builders to achieve this vision. Nucleus is designed to support and empower developers as they build on the Nillion Network.

Join the Nucleus Builders Program

The Genesis Cohort of Nucleus includes 12+ teams building self-sovereign AI data monetization, private order books, highly secure messaging and sharing apps, and many other interesting use cases. If you are a developer, team, or project interested in building a startup on the Nillion Network, apply to our Nucleus Builders Program.

Support

We provide comprehensive support across all key areas — technical, financial, marketing, and business to ensure that your journey from concept to production is smooth and successful.

Technical Support

We are committed to your success every step of the way. Nucleus offers weekly support sessions, where our technical experts work closely with you to guide the development of your project on the Nillion Network. Beyond these sessions, we provide daily communication through dedicated support channels, ensuring you have the resources and guidance needed to overcome any challenges and bring your product to life.

Financial Support

We believe in aligning our interests with those of our builders. We offer Nucleus builders a robust financial support package, including network credits that cover storage and compute costs on the Nillion Network. Additionally, our token compensation rewards provide tangible economic benefits as you build on our platform.

Marketing & Business Support

Your innovation deserves recognition. As part of Nucleus, we help raise awareness of your use cases through a range of marketing initiatives. From AMAs and Twitter Spaces to featured articles and more, we actively promote your projects to the broader community, helping you gain visibility and traction.

# permissions.md:

import DocCardList from '@theme/DocCardList';

Permissions

Grant other Nillion users permission to retrieve, update, or delete your secrets.

Setting permissions

Permissions examples demonstrated modifying the permissions of secrets. Check out the permissions folder on Github for examples of

- storing a permissioned secret
- retrieving a secret
- revoking permissions to read a secret
- checking that permissions were revoked

<DocCardList/>

# pynadac.md:

pynadac

The pynadac tool is a Nada compiler for Nada programs. It compiles programs to programname.nada.bin files and can also generate MIR JSON files like programname.nada.json.

Usage

```
bash
pynadac [OPTIONS] <PROGRAMPATH>
```

```
bash
Arguments:
  <PROGRAMPATH>  The path to the program to be compiled

Options:
  -t, --target-dir <TARGETDIR>  The target directory where output files will be
written to [default: .]
  -m, --generate-mir-json        Generate a MIR JSON file as an extra output
  -h, --help                     Print help
  -V, --version                  Print version
```

# python-client-examples.md:

import DocCardList from '@theme/DocCardList';

Python Client Examples

Nillion Python Starter Repo

The Nillion Python Starter Repo used in the Developer Quickstart contains single party compute examples, multi party compute examples, and examples involving secret permissions.

<DocCardList />

# python-client-reference.md:

Python Client Reference

Nillion Client Python module.

class pynillionclient.Amount

ByteSize()

Returns the size of the message in bytes.

Clear()

Clears the message.

ClearExtension()

Clears a message field.

ClearField()

Clears a message field.

CopyFrom()

Copies a protocol message into the current message.

DESCRIPTOR =

DiscardUnknownFields()

Discards the unknown fields.

Extensions

Extension dict

FindInitializationErrors()

Finds unset required fields.

FromString()

Creates new method instance from given serialized data.

HasExtension()

Checks if a message field is set.

HasField()

Checks if a message field is set.

IsInitialized()

Checks if all required fields of a protocol message are set.

ListFields()

Lists all set fields of a message.

MergeFrom()

Merges a protocol message into the current message.

MergeFromString()

Merges a serialized message into the current message.

ParseFromString()

Parses a serialized message into the current message.

static RegisterExtension(fielddescriptor)

SerializePartialToString()

Serializes the message to a string, even if it isn’t initialized.

SerializeToString()

Serializes the message to a string, only for initialized messages.

SetInParent()

Sets the has bit of the given field in its parent message.

UnknownFields()

Parse unknown field set

WhichOneof()

Returns the name of the field set inside a oneof, or None if no field is set.

class pynillionclient.Array(value)

This is a Array class used to
encode a secret array of elements.

Note: \\len\ method is implemented to allow
getting the length of the array.

Parameters:
value (list) – List of secret encoded elements.

Returns:
  Instance of the Array class.
Return type:
  Array
Raises:
  ValueError – invalid secret type: Raises an error when a public encoded element is included inside a
      secret array.

Example

```
py3
import pynillionclient as nillion

secretarray = nillion.Array([
    nillion.SecretInteger(1),
    nillion.SecretInteger(2),
])

print("The length of the array is: ", len(secretarray))
```

```
text
&gt;&gt;&gt; The length of the array is: 2
```

value

Getter method for the value inside a
Array instance.

Returns:
  List of secret encoded elements.
Return type:
  list

Example

```
py3
print("My secret array: \n", secretarray.value)
```

```
text
&gt;&gt;&gt; My secret array:
&gt;&gt;&gt;  [, ]
```

exception pynillionclient.AuthenticationError

Error related to authentication: invalid password, public key, or other internal errors

args

withtraceback()

Exception.withtraceback(tb) –
set self.\traceback\ to tb and return self.

class pynillionclient.ClusterDescriptor

The ClusterDescriptor contains attributes
that describe the cluster configuration. It includes information
about:

1. Cluster id;
2. Security parameters (statistical security and security threshold);
3. Parties (nodes) in the cluster;
4. Preprocessing configuration.

This object is returned when invoking NillionClient clusterinformation method.

Example

```
py3
cinfo = await client.clusterinformation(args.clusterid)
```

id

The Cluster identifier.

 Return type:
  A string containing the Nillion Cluster identifier.

Example

```
py3
cinfo = await client.clusterinformation(args.clusterid)
print("Cluster id:", cinfo.id)
```

```
text
&gt;&gt;&gt; Cluster id: 147f8d45-2126-4a54-9a64-8141ee55f51a
```

kappa

The statistical security parameter kappa for this cluster.

 Return type:
  The value of the statistical security parameter kappa used in this cluster.

Example

```
py3
cinfo = await client.clusterinformation(args.clusterid)
print("Statistical sec parameter kappa:", cinfo.kappa)
```

```
text
&gt;&gt;&gt; Statistical sec parameter kappa: 40
```

parties

Cluster’s parties ids.

 Return type:
  A list of strings containing the party identifiers in the cluster.

Example

py3
```
cinfo = await client.clusterinformation(args.clusterid)
print("Cluster parties' ids:", cinfo.parties)
```

text
```
&gt;&gt;&gt; Parties: {'12D3KooWJtRXjmV1HctQgvLUcrdxJ7cXwCHiL6PCheikN2rTJ2ZH',
              '12D3KooWHSveXS4DdXpCQyDDsp9D1x7fiTRnm1fsH9yJRpR6y4FM',
              '12D3KooWLV6HzUXpt6Tt5HUM5Fo3mpjvwsv9n4ADkJ962ArAZCvX'}
```

polynomialdegree

The polynomial degree used by this cluster. The polynomial
degree is directly related with the security threshold of the
Nillion network.

 Returns:
   An integer corresponding to the degree of the polynomial used
   in the cluster for linear secret sharing.

Example

py3
```
cinfo = await client.clusterinformation(args.clusterid)
print("Polynomial degree:", cinfo.polynomialdegree)
```

text
```
&gt;&gt;&gt; Polynomial degree: 1
```

prime

The prime number used in this cluster.

 Return type:
  The identifier of the prime used in the cluster.

Example

py3
```
cinfo = await client.clusterinformation(args.clusterid)
print("Prime type:", cinfo.prime)
```

text
```
&gt;&gt;&gt; Prime: U256SafePrime
```

exception pynillionclient.ComputeError

Error related to the computation: initialization, scheduling, or other internal errors

args

withtraceback()

Exception.withtraceback(tb) –
set self.\traceback\ to tb and return self.

class pynillionclient.ComputeFinishedEvent

The ComputeFinishedEvent class is
returned by an async computation when the computation
has just finished.

This class has no public constructor and is received from
method NillionClient.nextcomputeevent().

result

The computation’s result, as a FinalResult class.

Use the FinalResult.value() method to
obtain the wrapped value.

Returns:
: The FinalResult class containing the final result
  of the computation. Use the FinalResult.value() method to
  obtain the wrapped value.

Return type:
: FinalResult

Example

```
py3
uuid = await client.compute(
    args.clusterid,
    bindings,
    [storeid],
    args.computesecrets,
    args.computepublicvariables,
    paymentreceipt
)

while True:
    event = await client.nextcomputeevent()
    if isinstance(event, nillion.ComputeScheduledEvent):
        pass
    if isinstance(event, nillion.ComputeFinishedEvent) and event.uuid == uuid:
        print(
            f"Received computation result for {event.uuid}, result =
{event.result}"
        )
        print(
            f"Received computation result value for {event.uuid}, value =
{event.result.value}"
        )
        break
```

uuid

The computation’s UUID.

Returns:
: Uuid

Return type:
: str

Example

```
py3
uuid = await client.compute(
    args.clusterid,
    bindings,
    [storeid],
    args.computesecrets,
    args.computepublicvariables,
    paymentreceipt
)

while True:
    event = await client.nextcomputeevent()
    if isinstance(event, nillion.ComputeScheduledEvent):
        pass
    if isinstance(event, nillion.ComputeFinishedEvent) and event.uuid == uuid:
        print(
            f"Result for computation with UUID {event.uuid} is ready!"
        )
        break
```

class pynillionclient.ComputeScheduledEvent

The ComputeScheduledEvent class is
returned by an async computation when the computation is not finished yet.

This class has no public constructor and is received from
method NillionClient.nextcomputeevent().

uuid

The computation’s UUID. This outputs the same UUID
provided by the NillionClient.compute() method.

Returns:
: Computation UUID.

Return type:

  str

Example

```
py3
uuid = await client.compute(
    args.clusterid,
    bindings,
    [storeid],
    args.computesecrets,
    args.computepublicvariables,
    paymentreceipt
)

event = await client.nextcomputeevent()
if isinstance(event, nillion.ComputeScheduledEvent):
    computationuuid = event.uuid
```

class pynillionclient.ConnectionMode

This is a ConnectionMode class. It designates the
connection mode to use in a client constructor. We support three
different modes:

1. Dialer (dialer());
2. Direct (direct());
3. Relay (relay()).

dialer()

Specifies the client should connect in dialer mode.

In this mode the client only allows outgoing connections, so no need to
listen or open a port. This mode prohibits this client from recieving results,
as an output party, from the network.

 Return type:
  ConnectionMode

Example

```
py3
connectionmode = ConnectionMode.dialer()
```

direct()

Specifies a socket address structure for a listening client connection.

This mode is suited for clients that are backend services.

You allow incoming and outgoing connections, they are done directly so you have
to listen and have the port open to receive incoming connections. This option is
faster than relay as it avoids the extra hop, but requires to have the port open
/ have port forwarding in NATs and firewalls. This mode requires that the
client’s IPv4 address is addressible on the internet.

Parameters:
  str – Socket address structure.
Return type:
  ConnectionMode

Example

```
py3
connectionmode = ConnectionMode.direct('0.0.0.0:11337')
```

relay()

Specifies the client connects to the Nillion Network in
relay mode. So, if others want to contact the client, they
have to do so through a relay node that the client is connected to
(all nodes in the network are relay nodes).

This mode is suited for clients that cannot open a port like
phones behind a CGNAT, desktop apps behind a NAT or Firewall, and others.

You allow incoming and outgoing connection but the incoming are established via
a node in the network (the relayer) so that you don’t need to listen and open a
port.
This is slower than direct because it adds another hop in the connection but
allows to
bypass NATs and Firewalls. Also this option adds more load to the p2p network of
the
nodes becasue they have to do the relay increasing the incoming and outgoing
traffic
+ processing of secure connections.

Return type:
  ConnectionMode

Example

```
py3
connectionmode = ConnectionMode.relay()
```

exception pynillionclient.DealerError

Error related to the dealer: initialization, scheduling, unexpected errors

args

withtraceback()

Exception.withtraceback(tb) –
set self.\traceback\ to tb and return self.

class pynillionclient.FinalResult

This is a FinalResult class that is returned
from a finished computation.

This class has no public constructor and is received from
method ComputeFinishedEvent.result().

value

The resulting value of a computation.

 Returns:
  Result value from a computation.
 Return type:
  Dict

Example

```
py3
uuid = await client.compute(
    args.clusterid,
    bindings,
    [storeid],
    args.computesecrets,
    args.computepublicvariables,
    paymentreceipt
)

while True:
    event = await client.nextcomputeevent()
    if isinstance(event, nillion.ComputeScheduledEvent):
        pass
    if isinstance(event, nillion.ComputeFinishedEvent) and event.uuid == uuid:
        print(
            f"Received computation result value for {event.uuid}, value =
{event.result.value}"
        )
        break
```

class pynillionclient.Integer(value)

This is a Integer class used to
encode a public variable value as an integer.

Note: \\eq\ method is implemented to allow
to compare two integers.

 Parameters:
  value (int) – Value of the public encoded element.
 Returns:
  Instance of the Integer class.
 Return type:
  Integer

Example

```
py3
import pynillionclient as nillion

pubinteger1 = nillion.Integer(1)
```

```
pubinteger2 = nillion.Integer(2)

print("Are the public integers the same? ", pubinteger1 == pubinteger2)
```

```
text
&gt;&gt;&gt; Are the public integers the same?  False
```

value

Getter and setter for the value inside a
Integer instance.

 Returns:
  The value of the public integer.
 Return type:
  int

Example

```
py3
pubinteger = nillion.Integer(1)
print("Public integer is: ", pubinteger.value)
pubinteger.value = 2
print("Public integer is now: ", pubinteger.value)
```

```
text
&gt;&gt;&gt; Public integer is:  1
&gt;&gt;&gt; Public integer is now:  2
```

class pynillionclient.MsgPayFor

ByteSize()

Returns the size of the message in bytes.

Clear()

Clears the message.

ClearExtension()

Clears a message field.

ClearField()

Clears a message field.

CopyFrom()

Copies a protocol message into the current message.

DESCRIPTOR =

DiscardUnknownFields()

Discards the unknown fields.

Extensions

Extension dict

FindInitializationErrors()

Finds unset required fields.

FromString()

Creates new method instance from given serialized data.

HasExtension()

Checks if a message field is set.

HasField()

Checks if a message field is set.

IsInitialized()

Checks if all required fields of a protocol message are set.

ListFields()

Lists all set fields of a message.

MergeFrom()

Merges a protocol message into the current message.

MergeFromString()

Merges a serialized message into the current message.

ParseFromString()

Parses a serialized message into the current message.

static RegisterExtension(fielddescriptor)

SerializePartialToString()

Serializes the message to a string, even if it isn’t initialized.

SerializeToString()

Serializes the message to a string, only for initialized messages.

SetInParent()

Sets the has bit of the given field in its parent message.

UnknownFields()

Parse unknown field set

WhichOneof()

Returns the name of the field set inside a oneof, or None if no field is set.

class pynillionclient.NadaValues(values)

This is a NadaValues class used to
hold secrets and public values. It can contain:

1. Secret integers (SecretInteger);
2. Secret unsigned integers (SecretUnsignedInteger);
3. Arrays (Array).
4. Public integers (PublicInteger)
5. Public unsigned integers (PublicUnsignedInteger)

This class is used by the NillionClient.compute() method to pass the
secrets used by the corresponding Nada program that are not stored.

Parameters:
values (dict) – A map of named encoded secret and public values to store

Returns:

Instance of the NadaValues class.

Return type:
: NadaValues

Raises:
: ValueError – invalid public variable type: Raises an error when a public variabel element is included inside
      the secrets dictionary.

Example

```
py3
import pynillionclient as nillion

secuinteger = nillion.SecretUnsignedInteger(1)
secinteger = nillion.SecretInteger(1)
secarray = nillion.SecretArray([
    nillion.SecretInteger(1),
    nillion.SecretInteger(2),
])

secrets = nillion.NadaValues({
    "secuinteger": secuinteger,
    "secinteger": secinteger,
    "secarray": secarray
 })
```

dict()

Returns the stored values as a dictionary.

Returns:
: Native python dictionary with mapped encoded values

Return type:
: dict

Example

```
py3
values = nillion.NadaValues({
    "secuinteger": secuinteger,
    "secinteger": secinteger,
    "secarray": secarray
 })

print("Values:\n", values.dict())
```

```
text
&gt;&gt;&gt; Values:
&gt;&gt;&gt;  {'secarray': , 'secuinteger': , 'secinteger': }
```

class pynillionclient.NillionClient(nodekey, bootnodes, connectionmode, userkey, whitelist=None)

The NillionClient class serves as
the primary tool for connecting to the Nillion
network and facilitating various operations.
It allows users to interact with the Nillion

network efficiently, including for the following actions:

1. Store Nada programs (storeprogram());
2. Store values (storevalues());
3. Update values (updatevalues());
4. Retrieve values (retrievevalue());
5. Delete values (deletevalues());
6. Compute a Nada program over some secrets (compute());
7. Receive compute results (nextcomputeevent()).

An instance of NillionClient can embody either
a dealer node, responsible for providing inputs, or a result
node, tasked with receiving computation outputs. Under the hood,
this spawns a set of actors and acts as a node in the network that
has both dealer and result node capabilities.

Note: multiple instances can be employed concurrently if required;
however, it is imperative that each instance possesses
a distinct NodeKey when utilized within the
same interpreter.

Parameters:
- nodekey (NodeKey) – A private key to use for the Client node.
- bootnodes (list of str) – A list of nodes belonging to the network (other may be discovered later).
- connectionmode (ConnectionMode) – How to connect to the network, either directly (indicating a listen address), through a relay server or as a dialer client.
- userkey (UserKey) – User credentials to use.
- whitelist (list of str , optional) – A list of peer ids to connecto to/from.

Returns:
Instance of the NillionClient and an event receiver channel used to retrieve computation results.

Return type:
NillionClient

Example

For further information about the structure of the objects used by the
constructor, we refer to the quickstart guide .

```
py3
import pynillionclient as nillion

nodekey = nillion.NodeKey.fromfile("/path/to/node/key")
bootnodes = [os.getenv("NILLIONBOOTNODEMULTIADDRESS")]
e.g. bootnodes =
["/ip4/127.0.0.1/tcp/45305/p2p/11E4UiiRgsJILZYeushYEOQyMCrJLeRTaonNxBMBq4oF6bJ6M
foF"]
connectionmode = nillion.ConnectionMode.dialer()
userkey = nillion.UserKey.fromfile("/path/to/user/key")

client = nillion.NillionClient(
      nodekey,
      bootnodes,
      nillion.ConnectionMode.relay(),
      userkey,
  )
```

buildversion = 'client/0.1.0 (commit: 2454586480c93ea9664ddc563cac902e0bb03278;
ts: 1720014719; date: 2024-07-03T13:51:59+00:00)'

clusterinformation(clusterid)

Get information about a cluster by returning an instance of the ClusterDescriptor class.
We can access various information about the cluster through its methods.

Parameters:
  clusterid (str) – UUID of the targeted preprocessing cluster.
Return type:
  An instance of ClusterDescriptor populated with the cluster information.

Example

```
py3
await client.clusterinformation(clusterid)
```

compute(clusterid, bindings, storeids, values, receipt)

Requests a compute action in the Nillion Network for a specific Nada program under a set of secrets.

Note: This method does not directly output the result of the Nada program. Instead, it returns a computation UUID. To obtain the result, you’ll need to fetch it separately. The UUID, in conjunction with the event provided by the corresponding NillionClient instance channel, allows you to retrieve the computation results. Please refer to the e xample below for clarification.

Parameters:
  clusterid (str) – UUID of the targeted preprocessing cluster
  bindings (ProgramBindings) – The prepared program specification and secret bindings
  secrets (Secrets) – Additional secrets to use for the computation
  storeids (list of str) – List of the store IDs (uuids) of the secrets to use for the computation
  publicvariables (PublicVariables) – Public variables to use for the computation
  receipt (PaymentReceipt) – The receipt for the payment made.
Returns:
  A computation UUID.
Return type:
  str

Example

```
py3
import pynillionclient as nillion

storepaymentreceipt   = ... quote + pay
computepaymentreceipt = ... quote + pay

bindings = nillion.ProgramBindings(args.programid)
bindings.addinputparty("Dealer", client.partyid)
storeid = await client.storevalues(
    args.clusterid, bindings, args.storevalues, None, storepaymentreceipt
)

bindings = nillion.ProgramBindings(args.programid)
```

```
bindings.addinputparty("Dealer", client.partyid)
bindings.addoutputparty("Result", client.partyid)

uuid = await client.compute(
    args.clusterid,
    bindings,
    [storeid],
    args.values,
    computepaymentreceipt
)

while True:
    event = await client.nextcomputeevent()
    if isinstance(event, nillion.ComputeScheduledEvent):
        pass
    if isinstance(event, nillion.ComputeFinishedEvent) and event.uuid == uuid:
        print(
            f"Received computation result for {event.uuid}, result =
{event.result}"
        )
        print(
            f"Received computation result value for {event.uuid}, value =
{event.result.value}"
        )
        break
```

deletevalues(clusterid, storeid)

Delete existing values.

Parameters:
- clusterid (str) – UUID of the targeted preprocessing cluster
- storeid (str) – The identifier of the stored secret to be deleted (returned when calling storevalues())

Return type:
None

Example

```
py3
await client.deletevalues(clusterid, storeid)
```

nextcomputeevent()

Returns the state of the computation in the Nillion Network.

If the event is from an ongoing computation, it only includes
the corresponding UUID from the compute() process.
Once the computation is complete, the event includes both the
UUID and the computation result (FinalResult).

Returns:
Either event type will pull the next compute event from the internal result channel which can be inspected to determine if compute operation has completed

Return type:
ComputeScheduledEvent | ComputeFinishedEvent

Example

```
py3
uuid = await client.compute(
    args.clusterid,
    bindings,
    [storeid],
    args.computesecrets,
    args.computepublicvariables,
    paymentreceipt
)

while True:
    event = await client.nextcomputeevent()
    if isinstance(event, nillion.ComputeScheduledEvent):
        print(
            f"Waiting for computation with UUID={event.uuid} to finish."
        )
    if isinstance(event, nillion.ComputeFinishedEvent) and event.uuid == uuid:
        print(
            f"Received computation result for {event.uuid}, result =
{event.result}"
        )
        print(
            f"Received computation result value for {event.uuid}, value =
{event.result.value}"
        )
        break
```

partyid

Returns the SDK client’s instance party ID, which can be used
by the client to create program bindings (ProgramBindings,
check examples).

Effectively, the party ID is equivalent to the Peer ID
used within libp2p for inter-node communication. It is a hash
generated from the public key of the node’s key-pair (NodeKey). Not to
be confused with the userid() which is generated from the
public key of the user’s key-pair (UserKey).

Read more about party ID
and peer ID.

Returns:
 UUID of libp2p party identifier.
Return type:
 str

Example

```
py3
print("Party ID:", client.partyid)
```

requestpricequote(clusterid, operation)

Request a price quote for an operation. This method
asks the network to calculate a price quote for the

specified operation. Payment and submission of the
operation is the client’s responsibility and must be
done before the quote expires.

Note that the nodes of your target Nillion petnet cluster
will be bound to a single payment network (eg. testnet).

Parameters:
- clusterid (str) – UUID of the targeted preprocessing cluster
- operation (Operation) – The operation to get a price quote for.

Returns:
The price quoted for this operation.

Return type:
PriceQuote

Example

```
py3
secrets = pynillionclient.Secrets(
    {
         "foo": pynillionclient.SecretInteger(42),
         "bar": pynillionclient.SecretBlob(bytearray(b"hello world")),
    }
)
operation = pynillionclient.Operation.storevalues(secrets)
quote = await client.requestpricequote(clusterid, operation)
this is where you activate your payment method
txnhash = yourapp.yourpaymentmethodinunil(quote.cost)
paymentreceipt = nillion.PaymentReceipt(quote, txnhash)
storeid = await client.storevalues(
    clusterid, secrets, None, paymentreceipt
)
```

```
py3
updatedsecrets = nillion.Secrets({"foo": nillion.SecretInteger(42)})
operation = pynillionclient.Operation.updatevalues(updatedsecrets)
quote = await client.requestpricequote(clusterid, operation)
this is where you activate your payment method
txnhash = yourapp.yourpaymentmethodinunil(quote.cost)
paymentreceipt = nillion.PaymentReceipt(quote, txnhash)
 await client.updatevalues(
     args.clusterid, storeid, updatevalues, paymentreceipt
 )
```

```
py3
compute quote is based on compute time secrets; stored secrets have already
been paid
secrets = nillion.Secrets({"fortytwo": nillion.SecretInteger(42)})
quote = pynillionclient.Operation.compute(programid, secrets)
this is where you activate your payment method
txnhash = yourapp.yourpaymentmethodinunil(quote.cost)
paymentreceipt = nillion.PaymentReceipt(quote, txnhash)

uuid = await client.compute(
    args.clusterid,
    bindings,
    [storeid],
    secrets,
    pynillionclient.PublicVariables({}),
    computepaymentreceipt
)
```

```
py3
valuename = "fortytwo"
operation = pynillionclient.Operation.retrievevalue()
quote = await client.requestpricequote(clusterid, operation)
this is where you activate your payment method
txnhash = yourapp.yourpaymentmethodinunil(quote.cost)
paymentreceipt = nillion.PaymentReceipt(quote, txnhash)
result = await client.retrievevalue(clusterid, args.storeid, valuename,
paymentreceipt)
```

```
py3
programname = "myprogram"
programmirpath = f"./your/compiled/programs/{programname}.nada.bin"
operation = pynillionclient.Operation.storeprogram(programmirpath)
quote = await client.requestpricequote(clusterid, operation)
this is where you activate your payment method
txnhash = yourapp.yourpaymentmethodinunil(quote.cost)
paymentreceipt = nillion.PaymentReceipt(quote, txnhash)
programid = await client.storeprogram(
    args.clusterid, programname, programmirpath, paymentreceipt
)
```

```
py3
operation = pynillionclient.Operation.retrievepermissions()
quote = await client.requestpricequote(clusterid, operation)
this is where you activate your payment method
txnhash = yourapp.yourpaymentmethodinunil(quote.cost)
paymentreceipt = nillion.PaymentReceipt(quote, txnhash)
permissions = await client.retrievepermissions(clusterid, storeid,
paymentreceipt)
```

```
py3
operation = pynillionclient.Operation.updatepermissions()
quote = await client.requestpricequote(clusterid, operation)
this is where you activate your payment method
txnhash = yourapp.yourpaymentmethodinunil(quote.cost)
paymentreceipt = nillion.PaymentReceipt(quote, txnhash)
permissions = nillion.Permissions.defaultforuser(client.userid())
permissions.addretrievepermissions(set([args.retrieveruserid]))
updatedstoreid = await client.updatepermissions(args.clusterid, storeid,
permissions, paymentreceipt)
```

retrievepermissions(clusterid, storeid, receipt)

Retrieve permissions for a group of secrets in the Nillion Network

Parameters:
- clusterid (str) – UUID of the targeted preprocessing cluster
- storeid (str) – The secrets’ store ID (returned when calling storevalues())
- receipt (PaymentReceipt) – The receipt for the payment made.

Returns:
The permissions

Return type:
Permissions

Example

py3

```
permissions = await client.retrievepermissions(clusterid, storeid,
paymentreceipt)
```

retrievevalue(clusterid, storeid, valueid, receipt)

Retrieve a value stored in the Nillion Network.

To retrieve the value, you need to use the value
attribute on the second element of the output tuple.
Check the example below to read the value
of a secret integer.

Parameters:
- clusterid (str) – UUID of the targeted preprocessing cluster.
- storeid (str) – The value’s store ID (returned when calling storevalues()).
- valueid (str) – The value’s ID.
- receipt (PaymentReceipt) – The receipt for the payment made.

Returns:
The value ID as a UUID as well as the value itself.

Return type:
tuple

Example

```
py3
valuename = "fortytwo"
result = await client.retrievevalue(clusterid, args.storeid, valuename,
paymentreceipt)
print("Value ID: ", result[0])
print("Value: ", result[1])
```

```
text
&gt;&gt;&gt; Value ID: 2424a65c-d20c-4635-b864-06c064188dd4
&gt;&gt;&gt; Value: 42
```

storeprogram(clusterid, programname, programmirpath, receipt)

Store programs in the Nillion Network.

The programid used by storevalues() and compute() can be
built as follows:

```
py3
client.userid + "/" + programname
```

where client is a NillionClient instantiation and programname
is the name of the program.

Parameters:
- cluster. (\ clusterid - UUID of the targeted preprocessing) –
- store. (\ programname - Name of the program to) –
- stored. (\ programmirpath - Path to the MIR program being) –
- made. (\ receipt - The receipt for the payment) –

Returns:
The program identifier associated with the program

Return type:

```
  str

Example

py3
programname = "prog"
programmirpath = "programs-compiled/prog.nada.bin"

Store program in the Network
print(f"Storing program in the network: {programname}")
programid = await client.storeprogram(
    args.clusterid, programname, programmirpath, paymentreceipt
)
print("programid is: ", programid)




storevalues(clusterid, values, permissions, receipt)

Store values in the Nillion Network.

 Parameters:
   clusterid (str) – UUID of the targeted preprocessing cluster
   secrets (Secrets) – The secrets to store; this is a hash map indexed by
secret IDs
   permissions (Permissions , optional) – permissions to be set. By default the
user has update and retrieve permissions on the secret as well as compute
permissions for the program bound, should there be a program bound.
   receipt (PaymentReceipt) – The receipt for the payment made.
 Returns:
  A store identifier that can be used to retrieve the secret.
 Return type:
  str
 Raises:
  TypeError – When using bindings, the input party name provided (e.g.
“InputPartyName”) must
  match the input party name in the Nada program. Otherwise, we get a TypeError.

Example

Here are some examples of how to use this function. Note that to
give permissions we use the User ID and to bind a secret to a
program we use the Party ID.

py3
###########################
Example 1 - Simple      #
###########################
Notice that both bindings and permissions are set to None.
Bindings need to be set to use secrets in programs
Permissions need to be set to allow users other than the secret creator to use
the secret
secrets = nillion.Secrets({"fortytwo": nillion.SecretInteger(42)})
storeid = await client.storevalues(
    clusterid, secrets, None, paymentreceipt
)

###########################
Example 2 - Permissions #
###########################
permissions = nillion.Permissions.defaultforuser(client.userid)
permissions.addretrievepermissions(set([args.retrieveruserid]))
values = nillion.NadaValues({"fortytwo": nillion.SecretInteger(42)})
```

```
storeid = await client.storevalues(
    clusterid, secrets, permissions, paymentreceipt
)
```

updatepermissions(clusterid, storeid, permissions, receipt)

Update permissions for a group of secrets in the Nillion Network

Parameters:
- clusterid (str) – UUID of the targeted preprocessing cluster
- storeid (str) – The secrets’ store ID (returned when calling storevalues())
- permissions (Permissions , optional) – permissions to be set.
- receipt (PaymentReceipt) – The receipt for the payment made.

Returns:
The unique identifier of this update operation ID that can be used to help troubleshoot issues with this operation.

Return type:
str

Example

```
py3
Store
storepaymentreceipt = ... quote + pay for action
secrets = nillion.Secrets({"fortytwo": nillion.SecretInteger(42)})
storeid = await client.storevalues(
  clusterid, None, secrets, None, storepaymentreceipt
)

updatepaymentreceipt = ... quote + pay for action
Update permissions
permissions = nillion.Permissions.defaultforuser(client.userid())
permissions.addretrievepermissions(set([args.retrieveruserid]))
updatedstoreid = await client.updatepermissions(args.clusterid, storeid,
permissions, updatepaymentreceipt)

print("Stored secret id: ", storeid)
print("Updated stored secret id: ", updatedstoreid)
```

```
text
&gt;&gt;&gt; Stored secret id: 3c504263-fd3f-40b8-8a1d-9056b7846637
&gt;&gt;&gt; Updated stored secret id: ccdb8036-2635-40d9-9144-2cc89551fce9
```

updatevalues(clusterid, storeid, values, receipt)

Update values already stored in the Nillion Network.

Parameters:
- clusterid (str) – UUID of the targeted preprocessing cluster.
- storeid (str) – The secret’s store ID (returned when calling storevalues()).
- values (Secrets) – The values to update; this is a hash map indexed by secret IDs.
- receipt (PaymentReceipt) – The receipt for the payment made.

Returns:
The unique identifier of this update operation.

Return type:
str

Example

```
py3
updatedsecrets = nillion.Secrets({"fortytwo": nillion.SecretInteger(42)})
paymentreceipt = ... quote + pay
await client.updatevalues(
    args.clusterid, storeid, updatevalues, paymentreceipt
)
```

userid

Returns SDK client’s user ID, which is the public user
identifier.

The user ID is used to:

1. Generate a program ID (identification of a program in the Nillion Network).
Check example in storeprogram();
2. Grant a user permission to use secrets. Check Permissions.

It is a hash generated from the public key of the user’s key-pair (UserKey). Not to
be confused with the partyid() which is the generated from the
public key of the node’s key-pair (NodeKey).

Read more about user ID
in the Nillion Docs.

Returns:
Client’s user identifier.

Return type:
str

Example

```
py3
print("Party ID:", client.userid)
```

class pynillionclient.NodeKey

This is a NodeKey class that
contains a private key used by the
underlying libp2p to form multiaddress and
identity secrets. This class is consumed by NillionClient
class to initialize a client.

This object’s constructors can be used via the following
class methods:

1. From string encoded in Base58 (frombase58());
2. From a file (fromfile());
3. From a seed (fromseed()).

Example

```
py3
from pynillionclient import NodeKey
```

```
nodekey = NodeKey.fromseed('myseed')
```

frombase58()

Decodes a private key from a string encoded in Base58.

- Parameters:
  - contents (str) – A base58 string.
- Return type:
  - NodeKey

Example

```
py3
from pynillionclient import NodeKey
nodekey = NodeKey.frombase58()
```

fromfile()

Loads a file containing a private key.

- Parameters:
  - path (str) – The filesystem path to the file containing a base58 string.
- Return type:
  - NodeKey

Example

```
py3
from pynillionclient import NodeKey
nodekey = NodeKey.fromfile('/path/to/nodekey.base58')
```

fromseed()

Generates a private key using a seed.

- Parameters:
  - path (str) – A seed string.
- Return type:
  - NodeKey

Example

```
py3
from pynillionclient import NodeKey
nodekey = NodeKey.fromseed('myseed')
```

class pynillionclient.Operation

An operation that we want to run on the network.

compute(values)

Construct a new update values operation.

Parameters:
- programid (str) – The identifier of the program to be invoked.
- values (NadaValues) – The values to be stored.

Return type:
Operation

retrievepermissions()

Construct a new retrieve permissions operation.

Return type:
Operation

retrievevalue()

Construct a new retrieve value operation.

Return type:
Operation

storeprogram()

Construct a new store program operation.

Return type:
Operation

storevalues(ttldays)

Construct a new store values operation.

Parameters:
- values (Secrets) – The values to be stored.
- ttldays (int) – The time to live for the values in days

Return type:
Operation

updatepermissions()

Construct a new update permissions operation.

Return type:
Operation

updatevalues(ttldays)

Construct a new update values operation.

Parameters:
- values (NadaValues) – The values to be stored.
- ttldays (int) – The time to live for the values in days.

Return type:
Operation

exception pynillionclient.PaymentError

Payment-related errors: missing funds or other errors

args

withtraceback()

Exception.withtraceback(tb) –
set self.\traceback\ to tb and return self.

class pynillionclient.PaymentReceipt(quote, transactionhash)

A payment receipt.

Payment receipt are used to indicate that you made the payment for a price quote.

exception pynillionclient.PermissionError

Missing permission errors

args

withtraceback()

Exception.withtraceback(tb) –
set self.\traceback\ to tb and return self.

class pynillionclient.Permissions

This is a Permissions class used to
manage permissions of stored secrets and compute.
Permissions need to be set to allow users other
than the secret creator to use the secret. This
class is used either by NillionClient.storevalues()
or by NillionClient.updatepermissions().

The default instantiation of this class is given by
the method defaultforuser().

Example

See examples
on method requestpricequote().

addcomputepermissions(compute)

Add compute permissions to the Permissions instance for the
user IDs provided.

Parameters:
compute (dict of set of str) – Dict keyed by the userid of the targets where the value is a set of str
specifying which program IDs to permit compute for.

Example

```
py3
import pynillionclient as nillion

programid = client.userid() + "/" + "programname"
permissions = nillion.Permissions.defaultforuser(client.userid)
permissions.addcomputepermissions({
    args.computeuserid: {programid},
})
```

adddeletepermissions(delete)

Add delete permissions to the Permissions instance for the
given set of user IDs

Parameters:
delete (set of str) – Desired targets to permit delete Secrets.

Example

```
py3
import pynillionclient as nillion

permissions = nillion.Permissions.defaultforuser(client.userid)
permissions.adddeletepermissions(set([args.deleteuserid]))
```

addretrievepermissions(retrieve)

Add retrieve permissions to the Permissions instance for the
given set of user IDs.

Parameters:
retrieve (set of str) – Desired targets to permit read of stored programs or retrieve Secrets

Example

```
py3
import pynillionclient as nillion
```

```
permissions = nillion.Permissions.defaultforuser(client.userid)
permissions.addretrievepermissions(set([args.retrieveruserid]))
```

addupdatepermissions(update)

Add update permissions to the Permissions instance for the given set of user IDs.

Parameters:
  update (set of str) – Desired targets to permit update Secrets.

Example

```
py3
import pynillionclient as nillion

permissions = nillion.Permissions.defaultforuser(client.userid)
permissions.addupdatepermissions(set([args.updateuserid]))
```

static defaultforuser(userid)

Returns the default permission set for the given user ID.

Note: this method can be used to clear/revoke permissions previously granted by the user.

Parameters:
  userid (str) – Desired target user ID.
Return type:
  Permissions

Example

```
py3
import pynillionclient as nillion

permissions = nillion.Permissions.defaultforuser(client.userid)
```

iscomputeallowed(userid, program)

Returns true if user has compute permissions for every single program.

Return type:
  bool

Example

```
py3
programid = client.userid() + "/" + "programname"
computeallowed = permissions.iscomputeallowed(userclient.userid(), programid)
```

isdeleteallowed(userid)

Returns true if user has delete permissions.

 Return type:
  bool

Example

py3
```
import pynillionclient as nillion

permissions = nillion.Permissions.defaultforuser(client.userid)
deleteallowed = permissions.isdeleteallowed(userclient.userid)
print("Default user is always allowed: ", deleteallowed)
```

text
```
&gt;&gt;&gt; Default user is always allowed: True
```

isretrieveallowed(userid)

Returns true if user has retrieve permissions.

 Return type:
  bool

Example

py3
```
import pynillionclient as nillion

permissions = nillion.Permissions.defaultforuser(client.userid)
retrieveallowed = permissions.isretrieveallowed(userclient.userid)
print("Default user is always allowed: ", retrieveallowed)
```

text
```
&gt;&gt;&gt; Default user is always allowed: True
```

isretrievepermissionsallowed(userid)

Checks if user is allowed to retrieve the permissions.

 Return type:
  bool

Example

py3
```
retrievepermissionsallowed =
permissions.isretrievepermissionsallowed(userclient.userid())
```

isupdateallowed(userid)

Returns true if user has update permissions.

 Return type:
  bool

Example

```
py3
import pynillionclient as nillion

permissions = nillion.Permissions.defaultforuser(client.userid)
updateallowed = permissions.isupdateallowed(userclient.userid)
print("Default user is always allowed: ", updateallowed)
```

```
text
&gt;&gt;&gt; Default user is always allowed: True
```

isupdatepermissionsallowed(userid)

Checks if user is allowed to update the permissions.

 Return type:
  bool

Example

```
py3
updatepermissionsallowed =
permissions.isupdatepermissionsallowed(userclient.userid())
```

class pynillionclient.PriceQuote

A price quote for an operation to be run in the network.

Quotes can be requested by using NillionClient.requestpricequote().

cost

Gets the cost for the quoted operation in unil units.
The payment associated for the quoted operation must
transfer the total amount for it to be considered a valid
payment.

 Returns:
  The cost for this quote.
 Return type:
  OperationCost

Example

```
py3
print("Total Cost:", quote.cost.total)
```

expiresat

Gets the expiration time for this quote in seconds
since the unix epoch. The payment and the operation
execution must be invoked before this deadline is
hit, otherwise the network will reject the operation
request.

 Returns:
  The expiration time for this quote.
 Return type:
  int

Example

```
py3
print("Expiration time:", quote.expiresat)
```

nonce

Gets the nonce for this quote. This nonce must be used
as part of the payment transaction.

 Return type:
  bytearray

Example

```
py3
print("Nonce:", quote.nonce)
```

class pynillionclient.ProgramBindings(programid)

This is a ProgramBindings class used to
bind compute parties to explicit peer IDs (provided
by the NillionClient.partyid()). Bindings
need to be set to use secrets in programs

This class is used by the NillionClient.storevalues()
and NillionClient.compute() methods.

 Parameters:
  programid (str) – The identifier of the program to bind to.
 Returns:
  An instance of ProgramBindings.
 Return type:
  ProgramBindings

Example

```
py3
import pynillionclient as nillion

programid looks like: "/"
bindings = nillion.ProgramBindings(args.programid)

Add bindings when storing a secret
```

```
bindings.addinputparty("InputPartyName", client.partyid)
secrets = nillion.Secrets({"fortytwo": nillion.SecretInteger(42)})
storeid = await client.storevalues(
    clusterid, bindings, secrets, None
)
```

Add bindings for compute action

```
bindings = nillion.ProgramBindings(args.programid)
bindings.addinputparty("InputPartyName" client.partyid)
bindings.addoutputparty("OutputPartyName", client.partyid)
```

addinputparty(name, id)

Bind an input party with a name to a specific program.

Parameters:
- name (str) – The logical name of the input party in the Nada program.
- id (str) – The party identifier.

Example

py3

```
bindings = nillion.ProgramBindings(args.programid)
bindings.addinputparty("InputPartyName" client.partyid)
```

addoutputparty(name, id)

Bind an output party with a name to a specific program.

Parameters:
- name (str) – The name of the output party in the Nada program.
- id (str) – The party identifier.

Example

py3

```
bindings = nillion.ProgramBindings(args.programid)
bindings.addoutputparty("OutputPartyName", client.partyid)
```

exception pynillionclient.ProgramError

Program not found or invalid

args

withtraceback()

Exception.withtraceback(tb) –
set self.\traceback\ to tb and return self.

exception pynillionclient.ResultError

Errors related to fetching computation results

args

withtraceback()

Exception.withtraceback(tb) –
set self.\traceback\ to tb and return self.

class pynillionclient.SecretBlob(value)

This is a SecretBlob class used to
encode a secret as a blob.

Note: \\eq\ method is implemented to allow comparing two blobs.

 Parameters:
  value (bytearray) – Value of the secret blob as a bytearray.
 Returns:
  Instance of the SecretBlob class.
 Return type:
  SecretBlob
 Raises:
  VTypeError – argument ‘value’: Raises an error when a non-bytearray object is provided.

Example

```
py3
import pynillionclient as nillion

gmblobba = bytearray("gm, builder!", "utf-8")
gmblob = pynillionclient.SecretBlob(gmblobba)
readyblobba = bytearray("ready to build!", "utf-8")
readyblob = pynillionclient.SecretBlob(readyblobba)

print("Are these blobs the same?", gmblob == readyblob)
```

```
text
&gt;&gt;&gt; Are these blobs the same?  False
```

value

Getter and setter for the value inside a
SecretBlob instance.

 Returns:
  The value of the secret blob.
 Return type:
  int

Example

```
py3
gmblobba = bytearray("gm, builder!", "utf-8")
blob = nillion.SecretBlob(gmblobba)
print("Blob is: ", blob.value)
readyblobba = bytearray("ready to build!", "utf-8")
blob.value = readyblobba
print("Blob is now: ", blob.value)
```

```
text
&gt;&gt;&gt; Blob is:  bytearray(b'gm, builder!')
&gt;&gt;&gt; Blob is now:  bytearray(b'ready to build!')
```

class pynillionclient.SecretInteger(value)

This is a SecretInteger class used to
encode a secret as a integer.

Note: \\eq\ method is implemented to allow comparing two integers.

Parameters:
  value (int) – Value of the secret encoded element.
Returns:
  Instance of the SecretInteger class.
Return type:
  SecretInteger

Example

```
py3
import pynillionclient as nillion

secinteger1 = nillion.SecretInteger(1)
secinteger2 = nillion.SecretInteger(2)

print("Are the secret integers the same? ", secinteger1 == secinteger2)
```

```
text
&gt;&gt;&gt; Are the secret integers the same?  False
```

WARNING
Providing zero as SecretInteger leaks information.

value

Getter and setter for the value inside a
SecretInteger instance.

Returns:
  The value of the secret integer.
Return type:
  int

Example

```
py3
secinteger = nillion.SecretInteger(1)
print("Secret integer is: ", secinteger.value)
secinteger.value = 2
print("Secret integer is now: ", secinteger.value)
```

```
text
&gt;&gt;&gt; Secret integer is:  1
&gt;&gt;&gt; Secret integer is now:  2
```

class pynillionclient.SecretUnsignedInteger(value)

This is a SecretUnsignedInteger class used to
encode a secret as an unsigned integer.

Note: \\eq\ method is implemented to allow
comparing two unsigned integers.

Parameters:
 value (int) – Value of the secret encoded element.
Returns:
 Instance of the SecretUnsignedInteger class.
Return type:
 SecretUnsignedInteger
Raises:
 OverflowError – can’t convert negative int to unsigned: Raises an error when a negative integer value is used.

Example

```
py3
import pynillionclient as nillion

secuinteger1 = nillion.SecretUnsignedInteger(1)
secuinteger2 = nillion.SecretUnsignedInteger(2)

print("Are the secret unsigned integers the same? ", secuinteger1 ==
secuinteger2)
```

```
text
&gt;&gt;&gt; Are the secret unsigned integers the same?  False
```

WARNING
Providing zero as SecretUnsignedInteger leaks information.

value

Getter and setter for the value inside a
SecretUnsignedInteger instance.

Returns:
 The value of the secret unsigned integer.
Return type:
 int

Example

```
py3
secuinteger = nillion.SecretUnsignedInteger(1)
print("Secret unsigned integer is: ", secuinteger.value)
secuinteger.value = 2
print("Secret unsigned integer is now: ", secuinteger.value)
```

```
text
&gt;&gt;&gt; Secret unsigned integer is:  1
&gt;&gt;&gt; Secret unsigned integer is now:  2
```

exception pynillionclient.TimeoutError

Timed out

args

withtraceback()

Exception.withtraceback(tb) –
set self.\traceback\ to tb and return self.

class pynillionclient.UnsignedInteger(value)

This is a UnsignedInteger class used to
encode a public variable value as an unsigned integer.

Note: \\eq\ method is implemented to allow
to compare two unsigned integers.

 Parameters:
  value (int) – Value of the public encoded element.
 Returns:
  Instance of the UnsignedInteger class.
 Return type:
  UnsignedInteger
 Raises:
  OverflowError – can’t convert negative int to unsigned: Raises an error when a negative integer value is used.

Example

```
py3
import pynillionclient as nillion

pubuinteger1 = nillion.UnsignedInteger(1)
pubuinteger2 = nillion.UnsignedInteger(2)

print("Are the public unsigned integers the same? ", pubuinteger1 ==
pubuinteger2)
```

```
text
&gt;&gt;&gt; Are the public unsigned integers the same?  False
```

value

Getter and setter for the value inside a
UnsignedInteger instance.

Returns:
  The value of the public unsigned integer.

Return type:
  int

Example

```
py3
pubuinteger = nillion.UnsignedInteger(1)
print("Public unsigned integer is: ", pubuinteger.value)
pubuinteger.value = 2
print("Public unsigned integer is now: ", pubuinteger.value)
```

```
text
&gt;&gt;&gt; Public unsigned integer is:  1
&gt;&gt;&gt; Public unsigned integer is now:  2
```

class pynillionclient.UserKey

This is a UserKey class that
contains the public and private keys for the user.
This class is used by NillionClient
class to initialize a client.

This object’s constructors can be used via the following
class methods:

1. From string encoded in Base58 (frombase58());
2. From a file (fromfile());
3. From scratch (generate());
4. From seed (seed()).

frombase58()

Loads public and private key from base 58 encoded data.

Parameters:
  contents (str) – A base58 string.

Return type:
  UserKey

Example

```
py3
from pynillionclient import UserKey
userkey = UserKey.frombase58()
```

fromfile()

Loads public and private key from a file.

 Parameters:
  path (str) – The filesystem path to the file containing
  a base58 string.
 Return type:
  UserKey

Example

```
py3
from pynillionclient import UserKey
userkey = UserKey.fromfile('/path/to/userkey.base58')
```

fromseed()

Generates a public and private key from a seed.

 Parameters:
  seed (str) – A seed string.
 Return type:
  UserKey

Example

```
py3
from pynillionclient import UserKey
userkey = UserKey.fromseed('myseed')
```

generate()

Generates a random public and private key.

 Return type:
  UserKey

Example

```
py3
from pynillionclient import UserKey
userkey = UserKey.generate()
```

pynillionclient.createpaymentsmessage(quote: PriceQuote, payeraddress: str)

Create a payments message.

 Parameters:
   quote (PriceQuote) – The price quote for the operation being paid for.
   senderaddress (str) – The nilchain address of the payer.
 Returns:
  A protobuf message to be used when building a payments transaction.
 Return type:

  MsgPayFor

pynillionclient.version()

Return the version of this SDK client.

 Returns:
  The version of this build.
 Return type:
  str

Example

```
py3
pynillionclient.version()
```

# python-client.md:

```
import DocCardList from '@theme/DocCardList';
import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
```

Python Client

py-nillion-client is a Python client for building on top of the Nillion Network. It can be used to manage Nada programs, store and retrieve secrets, and run computations.

Installation

Install the py-nillion-client PyPi package in an existing Python application

```
bash
pip install --upgrade py-nillion-client
```

Usage

Import Python Client

```
python3
import pynillionclient as nillion
```

Initialize a client

Initialize an instance of the NillionClient connected to the Nillion Network using helpers from nillion-python-starter

```
<Tabs>
<TabItem value="main" label="main.py" default>
python
import os
from dotenv import loaddotenv pip install python-dotenv
from helpers.nillionclienthelper import createnillionclient
from helpers.nillionkeypathhelper import getUserKeyFromFile, getNodeKeyFromFile

Loads environment variables from the .env file
loaddotenv()
```

```
def main():
    userkey = getUserKeyFromFile(os.getenv("NILLIONUSERKEYPATHPARTY1"))
    nodekey = getNodeKeyFromFile(os.getenv("NILLIONNODEKEYPATHPARTY1"))
    Initialize Nillion Client instance
    client = createnillionclient(userkey, nodekey)
    Print the user id
    print(client.userid)

if name == "main":
    main()
```

</TabItem>

<TabItem value="client" label="nillionclienthelper.py">
python reference showGithubLink
https://github.com/NillionNetwork/nillion-python-starter/blob/main/helpers/nillionclienthelper.py

</TabItem>

<TabItem value="keypath" label="nillionkeypathhelper.py">
python reference showGithubLink
https://github.com/NillionNetwork/nillion-python-starter/blob/main/helpers/nillionkeypathhelper.py

</TabItem>

<TabItem value="env" label=".env">
python reference showGithubLink
https://github.com/NillionNetwork/nillion-python-starter/blob/main/.env.sample

</TabItem>
<TabItem value="bootstrap" label="bootstrapscript">
python reference showGithubLink
https://github.com/NillionNetwork/nillion-python-starter/blob/main/bootstrap-local-environment.sh

</TabItem>
</Tabs>
Resources

<DocCardList/>

# python-quickstart.md:

```
import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import IframeVideo from '@site/src/components/IframeVideo/index';
import SdkInstallation from './\sdk-installation.mdx';
```

Python Developer Quickstart

Welcome to the Python Quickstart. By the end of this guide, you will have:

1. Installed the Nillion SDK and set up your dev environment
2. Written, compiled, and tested your first nada program using the nada tool
3. Connected to the local devnet and run your program using the Python client

Once you have finished, explore more examples and tutorials in the Python examples repo to continue your Nillion developer journey!

Install the Nillion SDK tools

<SdkInstallation/>

Fork & clone the QuickStart repo and set up environment

The Nillion Python Starter repo has everything you need to start building.

1. Fork the repo on GitHub - it will be forked into a repo called your-github-username/nillion-python-starter
2. Clone the forked repo
```
   bash
   git clone https://github.com/<your-github-username>/nillion-python-starter.git
   cd nillion-python-starter
```

3. Ensure you have python3.11 or above:

   :::tip

   Use these commands to confirm that you have python3 (version >=3.11) and pip installed:

```
   bash
   python3 --version
   python3 -m pip --version
```

4. Create and activate a virtual environment
```
   bash
   python3 -m venv .venv
   source .venv/bin/activate
```

5. Intall the requirements
```
   bash
   pip install --upgrade -r requirements.txt
```

You now have everything we need to start your Nillion developer journey. We will work in the quickstart directory, however, if you ever get stuck, you can see a fully completed version of the quickstart in the quickstartcomplete directory.

Write your first nada program

The Nillion Network uses Nada, our MPC language, to define MPC programs. The first implementation of Nada is a Python DSL (Domain Specific Language), called Nada. In this section we will write a simple program that adds two numbers together.

Setup your project with the nada tool

Before we start writing your first nada program, we will use the nada tool to create our nada project which we will call nadaquickstartprograms.

```
bash
cd quickstart
nada init nadaquickstartprograms
```

This will create a directory called nadaquickstartprograms.

Your first program
The code for the finished program is below - it is a simple program that has one party and adds two secret integer inputs together.

```
python
from nadadsl import

def nadamain():

    party1 = Party(name="Party1")

    myint1 = SecretInteger(Input(name="myint1", party=party1))

    myint2 = SecretInteger(Input(name="myint2", party=party1))

    newint = myint1 + myint2

    return [Output(newint, "myoutput", party1)]
```

Now we will write it from scratch, explaining how it works as we go. Once we have written the program, we will use the nada tool to run and test it.

1. Create a program file:
   ```
   bash
   cd quickstart/nadaquickstartprograms/src
   touch secretaddition.py
   ```

2. Write or copy the program above into this file

Understanding the program you have just written

We will now go through the program slowly, and explain what is each part is doing.

1. First we must import nadadsl and create a function nadamain() - this function will contain our programs code.
   ```
   python
   from nadadsl import

   def nadamain():
   ```

2. Add a party
   ```
   python
   from nadadsl import

   def nadamain():

       party1 = Party(name="Party1")
   ```

   This is a one party program, however you can add multiple parties analogously, for example we could define party2 = Party(name="Party2").
3. Add inputs to the program
   ```
   python
   from nadadsl import

   def nadamain():

       party1 = Party(name="Party1")

       myint1 = SecretInteger(Input(name="myint1", party=party1))

       myint2 = SecretInteger(Input(name="myint2", party=party1))
   ```

   This program has two inputs, both secret integers. Each input must have a name and a party associated to it. Currently in nada you can only compute on secret or public integers (and rationals by using the nada-algebra library).

4. Compute on the inputs

```
   python
   from nadadsl import

   def nadamain():

       party1 = Party(name="Party1")

       myint1 = SecretInteger(Input(name="myint1", party=party1))

       myint2 = SecretInteger(Input(name="myint2", party=party1))

       newint = myint1 + myint2
```

   This performs a simple addition on the inputs. For all other built in operations available in nada, see here.

5. Return the output of the program

```
   python
   from nadadsl import

   def nadamain():

       party1 = Party(name="Party1")

       myint1 = SecretInteger(Input(name="myint1", party=party1))

       myint2 = SecretInteger(Input(name="myint2", party=party1))

       newint = myint1 + myint2

       return [Output(newint, "myoutput", party1)]
```

   To output the result of a program, we must provide a name - in this case myoutput - and a party to whom the output is provided - in this case party1.

Compile, run and test your program

Make sure you are in the quickstart/nadaquickstartprograms directory.

Now we will use the nada tool to compile, run and test the program we have just written. More information about the nada tool can be found here.

1. Add your program to nada-project.toml

   For the nada tool to know about our program, we need to add the following to the to the nada-project.toml file.

```
   bash
   [[programs]]
   path = "src/secretaddition.py"
   name = "secretaddition"
   primesize = 128
```

2. Build (compile) our program

```
   bash
   nada build
```

   This will compile all programs listed in the nada-project.toml file. You will see the binary files outputted in the nada-programs/target directory.

3. Generate test
   bash
   ```
   nada generate-test --test-name secretadditiontest secretaddition
   ```

   This uses the nada tool to generate a test, that will be stored in tests. Here secretadditiontest is the name of the test, and secretaddition is the name of the program we want to test. You will notice that the test file (tests/secretadditiontest.yaml) is automatically populated with 3s everywhere by default. Later, for the test to pass, we will have to change the output from 3 to the correct output.

4. Run the program
   bash
   ```
   nada run secretadditiontest
   ```

   Now we run the program. This uses the inputs defined in the test file (tests/secretadditiontest.yaml) and runs the program and prints the result. Make note of the result, we will need it next.

5. Test the program
   bash
   ```
   nada test secretadditiontest
   ```

   Finally, we test the program. If you run the above command without altering the default values (3s) in the test file (tests/secretadditiontest.yaml), the test will fail.

Connect to the devnet and run your program
We have written and tested our nada program, now we need to run it against the local devnet. In this section we will:

1. Spin up a local Nillion devnet.
2. Use the Nillion python client to interact with the local devnet and compute the program we have just written.

Spinning up a local Nillion devnet
Spinning up a local Nillion devnet is easy, simply run the following command:

bash
```
nillion-devnet
```

All configurations of the devnet you will need are written to the following environment file /Users/<user>/.config/nillion/nillion-devnet.env
You need to leave the devnet running in the background while you run your program in the next section.

Using the Python client to run your program

In this section, we will use the python client run a computation on the local devnet.

We will write the following code within the quickstart/clientcode directory in the runmyfirstprogram.py file here. You can view the completed client code here, feel free to refer back to it whenever you need.

1. Import the packages and helper functions we will be using

   python
   ```
   import asyncio
   import pynillionclient as nillion
   ```

```
   import os

   from pynillionclient import NodeKey, UserKey
   from dotenv import loaddotenv
   from nillionpythonhelpers import getquoteandpay, createnillionclient,
createpaymentsconfig

   from cosmpy.aerial.client import LedgerClient
   from cosmpy.aerial.wallet import LocalWallet
   from cosmpy.crypto.keypairs import PrivateKey

   home = os.getenv("HOME")
   loaddotenv(f"{home}/.config/nillion/nillion-devnet.env")

   async def main():
```

Here the cosmpy imports will help us interact with the local chain, the helper functions help abstract away some of the technical details when using the python client, and finally we load the .env file containing the configs of the local devnet.

2. Obtain the local devnet configs and create a user & node key from a seed

```
    python
    1. Initial setup
    1.1. Get clusterid, grpcendpoint, & chainid from the .env file
    clusterid = os.getenv("NILLIONCLUSTERID")
    grpcendpoint = os.getenv("NILLIONNILCHAINGRPC")
    chainid = os.getenv("NILLIONNILCHAINCHAINID")
    1.2 pick a seed and generate user and node keys
    seed = "myseed"
    userkey = UserKey.fromseed(seed)
    nodekey = NodeKey.fromseed(seed)
```

Here we first obtain the clusterid, grpcendpoint & chainid from the local environment. Then we choose a seed and obtain a user and node key using the fromseed method.

3. Initialise a Nillion client & obtain user and party ids

```
    python
    2. Initialize NillionClient against nillion-devnet
    Create Nillion Client for user
    client = createnillionclient(userkey, nodekey)

    partyid = client.partyid
    userid = client.userid
```

Here we use the createnillionclient helper to create the client that will act on behalf of the party and obtain the party and user ids which identify the party.

4. Pay for and store a program

```
    python
    3. Pay for and store the program
    Set the program name and path to the compiled program
    programname = "secretaddition"
    programmirpath = f"../nadaquickstartprograms/target/{programname}.nada.bin"

    Create payments config, client and wallet
    paymentsconfig = createpaymentsconfig(chainid, grpcendpoint)
    paymentsclient = LedgerClient(paymentsconfig)
    paymentswallet = LocalWallet(
        PrivateKey(bytes.fromhex(os.getenv("NILLIONNILCHAINPRIVATEKEY0"))),
        prefix="nillion",
    )

    Pay to store the program and obtain a receipt of the payment
    receiptstoreprogram = await getquoteandpay(
        client,
        nillion.Operation.storeprogram(programmirpath),
        paymentswallet,
        paymentsclient,
        clusterid,
    )

    Store the program
    actionid = await client.storeprogram(
        clusterid, programname, programmirpath, receiptstoreprogram
    )

    Create a variable for the programid, which is the {userid}/{programname}. We
will need this later
    programid = f"{userid}/{programname}"
    print("Stored program. actionid:", actionid)
    print("Stored programid:", programid)
```

We first construct the path to the compiled program. Then we create the payments config, client and wallet - we use cosmpy to do this along with a number of parameters of the devnet. Next we use the pay helper function to pay for storing the program - you will see the operation (storeprogram) is an input parameter. When this function is called, a quote for storing the program is asked for and received before the payment is made. Look at the pay function here to understand the precise flow in more detail. Finally we store the program (ensuring we provide a valid receipt) and then construct the programid as we will need this later. Note: program ids always follow the same structure.

5. Pay for and store a secret

```
    python
    4. Create the 1st secret, add permissions, pay for and store it in the
network
    Create a secret named "myint1" with any value, ex: 500
    newsecret = nillion.NadaValues(
        {
            "myint1": nillion.SecretInteger(500),
        }
    )

    Set the input party for the secret
    The party name needs to match the party name that is storing "myint1" in the
program
    partyname = "Party1"

    Set permissions for the client to compute on the program
    permissions = nillion.Permissions.defaultforuser(client.userid)
    permissions.addcomputepermissions({client.userid: {programid}})

    Pay for and store the secret in the network and print the returned storeid
    receiptstore = await getquoteandpay(
        client,
        nillion.Operation.storevalues(newsecret, ttldays=5),
        paymentswallet,
        paymentsclient,
        clusterid,
    )
    Store a secret
    storeid = await client.storevalues(
        clusterid, newsecret, permissions, receiptstore
    )
    print(f"Computing using program {programid}")
    print(f"Use secret storeid: {storeid}")
```

First we create a secret object, making sure the name of the secret (myint1) matches the name of the secret in the program. Then we create compute permissions; even if a party is computing on its own secret it still needs to grant permissions. Finally we pay for the storage and obtain a receipt, and finally we pass the receipt and permissions to the storevalues method which stores the secret in the network.

6. Pay for and action the computation

```
    python
    5. Create compute bindings to set input and output parties, add a
computation time secret and pay for & run the computation
    computebindings = nillion.ProgramBindings(programid)
    computebindings.addinputparty(partyname, partyid)
    computebindings.addoutputparty(partyname, partyid)

    Add myint2, the 2nd secret at computation time
    computationtimesecrets = nillion.NadaValues({"myint2":
nillion.SecretInteger(10)})

    Pay for the compute
    receiptcompute = await getquoteandpay(
        client,
        nillion.Operation.compute(programid, computationtimesecrets),
        paymentswallet,
        paymentsclient,
        clusterid,
    )

    Compute on the secret
    computeid = await client.compute(
        clusterid,
        computebindings,
        [storeid],
        computationtimesecrets,
        receiptcompute,
    )
```

Before running a computation, we have to create bindings which set the input and output parties for the program - in this case, this is all the same party, Party1. We then add a computation time secret which will act as the second input to the program (myint2). As above we then pay for an run the computation. Note that we must provide the programid and computationtimesecrets to the pay function, this is so a quote can be correctly generated. When running the computation, the receipt is checked to ensure it is valid for that particular computation and the provided inputs.

7. Return the result of the computation

```
    python
    6. Return the computation result
    print(f"The computation was sent to the network. computeid: {computeid}")
    while True:
        computeevent = await client.nextcomputeevent()
        if isinstance(computeevent, nillion.ComputeFinishedEvent):
            print(f"✅  Compute complete for computeid {computeevent.uuid}")
            print(f"🖥️  The result is {computeevent.result.value}")
            return computeevent.result.value
```

Finally we return the result of the computation. Here we await for the next event to be available in the network, and then print the result.

8. Run the completed python script

   Ensure you can run the script by putting the following at the end:

```
   python
   if name == "main":
       asyncio.run(main())
```

   Run the script to store the program, store secrets and compute on the program.

```
   bash
   cd clientcode
   mv runmyfirstprogram.py secretaddition.py
   python3 secretaddition.py
```

   You will now see the program executing and the result printed.

Keep exploring

Congratulations, you've successfully written your first single party Nada program and run it on the local devnet. Checkout the resources below to continue your Nillion developer journey.

- Python examples repo:
  - Examples:
    - single party examples
    - multi-party examples
    - storing and retrieving intergers and blobs
    - using permissions
  - Tutorials:
    - Voting schemes
    - Millionaire's tutorial
      <IframeVideo videoSrc="https://www.loom.com/embed/d77604f001be4293b1b0c72c67620071?sid=d8dba7d7-0643-47cf-bf44-8b8b33c18cd6"/>

# quickstart-blind-app.md:

```
import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import IframeVideo from '@site/src/components/IframeVideo/index';
import SdkInstallation from './\sdk-installation.mdx';
import QuickstartIntro from './\quickstart-intro.mdx';
import VenvSetup from './\nada-venv-setup.mdx';
import UnderstandingProgram from './\understanding-first-nada-program.mdx';
import CompileRunTest from './\quickstart-compile-run-test.mdx';
```

Build a Blind App with the cra-nillion starter repo

:::info

This is the 3rd step in the Blind App Quickstart. Before starting this guide,

1. Install the Nillion SDK
2. Create a Nada project and write your first Nada program

:::

The cra-nillion Starter Repo repo is a Create React App which has everything you need to start building your blind app.

Clone the CRA-Nillion JavaScript starter repo

Make sure you are in the root of the quickstart directory. Clone the repo into your quickstart directory.

```
bash
git clone https://github.com/NillionNetwork/cra-nillion.git
```

Install repo dependencies and run the starter

:::info

Before you use cra-nillion, check that you have Node (>= v18.17) installed by running
node -v
:::

```
cd cra-nillion
npm i
npm start
```

Open http://localhost:8080/ to see your cra-nillion starter app running locally at port 8080

!CRA nillion no cluster

For this Quickstart, we'll focus on the Nillion Operations page and the Nillion Blind Computation Demo page.

Connect the blind app to nillion-devnet

In the screenshot of cra-nillion, you'll notice that cluster id and other configuration variables needed to connect to the Nillion Network are not set, so it's not possible to connect NillionClient.

Spin up a local Nillion devnet

Open a second terminal and run the devnet using any seed (the example uses "my-seed") so the cluster id, websockets, and other environment variables stay constant even when you restart nillion-devnet.

```
shell
nillion-devnet --seed my-seed
```

You will see an output like this:

```
nillion-devnet --seed my-seed
```

```
ℹ️ cluster id is 222257f5-f3ce-4b80-bdbc-0a51f6050996
ℹ️ using 256 bit prime
ℹ️ storing state in /var/folders/1/2yw8krkx5q5dn2jbhx69s4r0000gn/T/.tmpU00Jbm
(62.14Gbs available)
🏃 starting nilchain node in:
/var/folders/1/2yw8krkx5q5dn2jbhx69s4r0000gn/T/.tmpU00Jbm/nillion-chain
⛓  nilchain JSON RPC available at http://127.0.0.1:48102
⛓  nilchain gRPC available at localhost:26649
🏃 starting node 12D3KooWMGxv3uv4QrGFF7bbzxmTJThbtiZkHXAgo3nVrMutz6QN
⌛ waiting until bootnode is up...
🏃 starting node 12D3KooWKkbCcG2ujvJhHe5AiXznS9iFmzzy1jRgUTJEhk4vjF7q
🏃 starting node 12D3KooWMgLTrRAtP9HcUYTtsZNf27z5uKt3xJKXsSS2ohhPGnAm
💰 funding nilchain keys
📝 nillion CLI configuration written to /Users/steph/Library/Application
Support/nillion.nillion/config.yaml
🌄 environment file written to /Users/steph/Library/Application
Support/nillion.nillion/nillion-devnet.env
```

Copy the path printed after "🌄 environment file written to" and open the file

```
vim "/Users/steph/Library/Application Support/nillion.nillion/nillion-
devnet.env"
```

This file has the nillion-devnet generated values for cluster id, websocket, json rpc, and private key. You'll need to put these in your local .env in one of the next steps so that your cra-nillion demo app connects to the nillion-devnet.

Keep the nillion-devnet running in this terminal.

Create .env file

Make sure you are in the quickstart/cra-nillion directory.

Copy the up the .env.example file to a new .env and set up these variables to match the nillion environment file.

```
shell
cp .env.example .env
```

Update your newly created .env with environment variables outout in your terminal by nillion-devnet

```
REACTAPPNILLIONCLUSTERID=
REACTAPPNILLIONBOOTNODEWEBSOCKET=
REACTAPPNILLIONNILCHAINJSONRPC=
REACTAPPNILLIONNILCHAINPRIVATEKEY=
REACTAPPAPIBASEPATH=/nilchain-proxy

Optional: add your ETH Address to enable JavaScript Client Telemetry
REACTAPPYOURETHEREUMADDRESSFORNILLIONTELEMETRY=
```

Restart the cra-nillion app process

```
npm start
```

Now the Cluster ID field should be populated with the nillion-devnet cluster id value you set in REACTAPPNILLIONCLUSTERID.

!CRA nillion with cluster

Try out the Operations Page

1. Generate User Key - generates a new user key / user id pair
2. Connect with User Key - sets the user key and connects to NillionClient via the Nillion JavaScript Client
3. Hide Nillion User Key and Node Key Connection Info - toggle button to show/hide user and node key options
4. Perform Nillion Operations

To perform an operation (store secret, retrieve secret, update secret, store program, compute), you follow the same pattern:

1. Get quote for the operation
2. Pay quote for the operation and get a payment receipt. On your local nillion-devnet, payments for operations are sent to the local nilchain at REACTAPPNILLIONNILCHAINJSONRPC are funded by REACTAPPNILLIONNILCHAINPRIVATEKEY
3. Perform the operation with the payment receipt as a parameter

   !CRA nillion operations

Hook up your secretaddition.py nada program to your first blind app

Now that you understand how Nillion operations work, let's update the Nillion Blind Computation Demo page to use the Nada program you created.

Navigate to the Blind Computation Demo page: http://localhost:8080/compute

The code for Blind Computation Demo page lives in ComputePage.tsx

```
const outputName = 'myoutput';
const partyName = 'Party1';
```

Notice that the ComputePage sets outputName which matches the the output name set in secretaddition.py. The ComputePage sets partyName which matches the the party name set in secretaddition.py. There are 2 StoreSecretForm components on ComputePage, with secretName set to myint1 and myint2 which matches the the secret names set in secretaddition.py.

<Tabs>
<TabItem value="helpers" label="secretaddition.py" default>

```
from nadadsl import

def nadamain():

    party1 = Party(name="Party1")

    myint1 = SecretInteger(Input(name="myint1", party=party1))

    myint2 = SecretInteger(Input(name="myint2", party=party1))

    newint = myint1 + myint2

    return [Output(newint, "myoutput", party1)]
```

</TabItem>

<TabItem value="compute" label="ComputePage.tsx" >
ts reference showGithubLink
https://github.com/NillionNetwork/cra-nillion/blob/main/src/ComputePage.tsx

</TabItem>

</Tabs>

Update programName

In order to run blind computation on your secretaddition.py Nada program, you'll need to make a few updates:

1. Open a new terminal and navigate to the root quickstart folder. List the contents of the folder

```
ls
```

You should see cra-nillion and nadaquickstartprograms folders.

2. Copy your secretaddition.py and secretaddition.nada.bin files nadaquickstartprograms into cra-nillion

```
cp nadaquickstartprograms/src/secretaddition.py cra-nillion/public/programs
cp nadaquickstartprograms/target/secretaddition.nada.bin
cra-nillion/public/programs
```

Now your cra-nillion app can use the nada program and the nada program binaries in store program operations.

3. Update programName to secretaddition so the cra-nillion repo reads your Nada program.

```
ts
// const programName = 'additionsimple'; <-- Change the string
const programName = "secretaddition";
```

:::tip

Open the Nillion JavaScript Client Reference doc in another tab to search for available Nillion Client classes while working with cra-nillion.

:::

Run the Blind Computation Demo

Go back to the Blind App on http://localhost:8080/compute and run through the steps on the page to test out the full blind computation flow.

Next steps

🚀 Woohoo! You've successfully built your first local blind app by writing a Nada program, storing the program on the network, storing secrets on the network, and running compute against secrets. Next, deploy your blind app to the Nillion Network Testnet so anyone in the world can try it.

# quickstart-install.md:

import SdkInstallation from './\sdk-installation.mdx';

Install Nillion

Install the Nillion SDK, including the <strong>nada</strong> tool you'll use to create a Nada project and the <strong>nillion-devnet</strong> tool you'll use to spin up a local Nillion network.

Use nilup to install the Nillion SDK

<SdkInstallation/>

Next steps

✅ You're off to a great start! Now that you have installed the Nillion SDK and Tools, you have everything you need to create a Nada project.

# quickstart-nada.md:

import VenvSetup from './\nada-venv-setup.mdx';
import UnderstandingProgram from './\understanding-first-nada-program.mdx';
import CompileRunTest from './\quickstart-compile-run-test.mdx';
import PythonVersionInfo from './\python-version-info.mdx';

Create a Nada project and write your first Nada program

The <strong>nada</strong> tool is used to create a new Nada project, compile all programs in the project, run Nada programs, and generate tests for Nada programs.

:::info

Before starting this guide, Install the Nillion SDK

Confirm installation:

```
nillion -V
```

:::

Create a new Nada project

Create a quickstart directory. Inside of quickstart, use the <strong>nada</strong> tool to create a new nada project named "nadaquickstartprograms" by running

```
mkdir quickstart
cd quickstart
nada init nadaquickstartprograms
```

Set up a Python virtual environment and install nada-dsl

The Nillion Network leverages Nada, our MPC language, for defining MPC programs. Our initial implementation of Nada comes in the form of Nada, a Python DSL (Domain Specific Language).

<PythonVersionInfo/>

0. Change directories into your new nada project directory

```
cd nadaquickstartprograms
```

<VenvSetup/>

Write your first Nada program

Open the nadaquickstartprograms folder in your code editor of choice.

:::info

Your Nada project has the following structure:

```
nada-project.toml  Configuration file for your Nada programs
src/               Directory for writing Nada programs
src/main.py        An example Nada program that adds 2 secret integers
target/            Directory where nada build puts resulting compiled programs
tests/             Directory where nada generate-test puts resulting test files
```

:::

1. Create a Nada program file

The src/ directory is where you'll write your Nada programs. Create a new Nada program file in the src directory. The secretaddition.py Nada program will add 2 SecretInteger Nada values together.

```
touch src/secretaddition.py
```

2. Write the secret addition Nada program

Copy this Nada program into the secretaddition.py file. Don't worry, we'll go through a line by line explanation of the program next.

```
python
from nadadsl import

def nadamain():

    party1 = Party(name="Party1")

    myint1 = SecretInteger(Input(name="myint1", party=party1))

    myint2 = SecretInteger(Input(name="myint2", party=party1))

    newint = myint1 + myint2

    return [Output(newint, "myoutput", party1)]
```

3. Understand each line of the secretaddition.py Nada program

<UnderstandingProgram/>

Compile, run, and test your Nada program

<CompileRunTest/>

Next steps

▦ Fantastic work! You just created your first Nada project and wrote and tested a Nada program. Now you can move onto the final step: hooking up the secretaddition Nada program to a blind app, which will be able to store the program, then compute with the program on secret, user provided inputs.

# quickstart-testnet.md:

```
import JsHeaders from './\js-headers-proxy.mdx';
import IframeVideo from '@site/src/components/IframeVideo/index';
import {ReactTestnetEnv} from '@site/src/components/Networks/TestnetEnv';
```

Deploy your blind app to the Nillion Testnet

Your blind app is currently running locally against the nillion-devnet. Let's configure environment variables to point at the Nillion Testnet. That way anyone can play with your blind app once it's deployed.

Update your .env file and test locally

Update your .env values to point at the Nillion Testnet

<ReactTestnetEnv/>

The REACTAPPNILLIONNILCHAINPRIVATEKEY private key value above should correspond to an address you've funded with Testnet NIL.

Create a Nillion Wallet and get the private Key

Follow the Creating a Nillion Wallet guide to create your Nillion wallet. Note that when you create your wallet, you need to use the "Sign up with Google" option rather than "Use recovery phrase" option because Keplr only exposes the private key of wallets created when you "Sign up with Google."

Here's how to get the REACTAPPNILLIONNILCHAINPRIVATEKEY value from a Nillion wallet created with a Google account:

<IframeVideo
videoSrc="https://www.loom.com/embed/0f9949a990ee497195a39e02b280f2c7?sid=f53b62d2-6820-4780-98b1-5b3049b00709"/>

Fund the Nillion Wallet address that corresponds to your private key

Follow the Nillion Faucet Guide to learn how to get Testnet NIL to fund the Nillion wallet address that corresponds to your private key. This way your app can pay for operations.

Test the configuration locally against your blind app to make sure the full blind computation flow is working as expected.

Set Headers and set up proxy for nilchain

<JsHeaders/>

Commit your project to Github

Commit your repo to Github and tag your Github repo with nillion-nada so the rest of the Nillion community can find it.

Host your blind app with Vercel

1. Follow the https://vercel.com/docs/getting-started-with-vercel/import guide to import your Github project to Vercel

2. Follow the https://vercel.com/docs/projects/environment-variables guide to add all Testnet environment variables

3. Set up the vercel.json file with headers and proxy rewrites

json reference showGithubLink
https://github.com/NillionNetwork/cra-nillion/blob/main/vercel.json

4. Share your live link on Nillion's Github Discussions Show and Tell Forum

Keep Exploring

🥳 Congratulations on completing the Nillion Developer Quickstart and deploying your blind app to the Nillion testnet. Keep exploring and building by

- reading about Nillion concepts and the Nada Language
- learning how to interact with and manage programs, secrets, and permissions on the Nillion Network with Nillion Client
- challenging yourself to create a page that solves the Millionaires Problem

# quickstart.md:

```
import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import IframeVideo from '@site/src/components/IframeVideo/index';
import LinkButton from '@site/src/components/LinkButton/index';
import SdkInstallation from './\sdk-installation.mdx';
```

Blind App Quickstart

What is a blind app?

A blind app runs blind computation on Nillion using one or more privacy-preserving Nada programs. Nada programs compute on secret integers without ever seeing the underlying input values, making them ideal for sensitive data operations.

Start building blind apps on Nillion

:::info
This Blind App Quickstart is ideal for developers interested in creating frontends or fullstack web apps on Nillion. If you want to connect a backend to Nillion, check out the Python Quickstart
:::

Build your first blind application on Nillion. Follow the steps in this blind app quickstart to

1. Install the Nillion SDK
2. Create a Nada project and write your first Nada program
3. Build a blind app with the cra-nillion starter repo (based on Create React App) and run your app locally on the nillion-devnet
4. Deploy your blind app to the Nillion Testnet so the world can try it

# retrieve-secret-js.md:

```
import Tabs from '@theme/Tabs';
```

import TabItem from '@theme/TabItem';

Retrieve Value

Retrieve secret strings and integers from the network. Retrieve and decode a stored secret string (SecretBlob) or a stored secret integer (SecretInteger)

Get a Quote to Retrieve Value

<Tabs>

<TabItem value="getquote" label="Get quote to retrieve secret" default>
ts reference showGithubLink
https://github.com/NillionNetwork/cra-nillion/blob/main/src/nillion/components/RetrieveSecretForm.tsx#L50-L55

</TabItem>

<TabItem value="getQuote" label="getQuote helper">
ts reference showGithubLink
https://github.com/NillionNetwork/cra-nillion/blob/main/src/nillion/helpers/getQuote.ts

</TabItem>
</Tabs>

Pay to Retrieve Value and get Payment Receipt

<Tabs>

<TabItem value="receipt" label="Payment receipt" default>
ts reference showGithubLink
https://github.com/NillionNetwork/cra-nillion/blob/main/src/nillion/components/RetrieveSecretForm.tsx#L69-L76

</TabItem>

<TabItem value="helpers" label="helper functions">
ts reference showGithubLink
https://github.com/NillionNetwork/cra-nillion/blob/main/src/nillion/helpers/nillion.ts#L24-L71

</TabItem>

</Tabs>

Retrieve SecretBlob or SecretInteger

<Tabs>

<TabItem value="receipt" label="Retrieve value" default>
ts reference showGithubLink
https://github.com/NillionNetwork/cra-nillion/blob/main/src/nillion/components/RetrieveSecretForm.tsx#L79-L84

</TabItem>

<TabItem value="retrieve" label="retrieveSecret helper">
ts reference showGithubLink
https://github.com/NillionNetwork/cra-nillion/blob/main/src/nillion/helpers/retrieveSecret.ts

</TabItem>

</Tabs>

# retrieve-secret.md:

Retrieve Secret

Retrieve secret strings, integers, and arrays from the network.

retrievesecret returns a tuple containing two elements: the first element is the name of the secret represented as a UUID (Universally Unique Identifier), and the second element is the actual secret value.

:::tip

Open the Nillion Python Client Reference in another tab to search for available Nillion Client classes.

:::

Retrieve a SecretBlob

Get a quote to retrieve a value, pay to retrieve a value, get a payment receipt, and finally retrieve the SecretBlob

python reference showGithubLink
https://github.com/NillionNetwork/python-examples/tree/main/examplesandtutorials/coreconceptstoreandretrievesecrets/storeandretrieveblob.py#L70-L89

Retrieve a SecretInteger

Get a quote to retrieve a value, pay to retrieve a value, get a payment receipt, and finally retrieve the SecretInteger

python reference showGithubLink
https://github.com/NillionNetwork/python-examples/tree/main/examplesandtutorials/coreconceptstoreandretrievesecrets/storeandretrieveinteger.py#L68-L85

# sdk-instrumentation.md:

SDK Instrumentation

Nillion SDK collects telemetry data to understand how the software is used and to better assist you in case of issues.
This telemetry is opt-in, by default we don't collect any data. If you want to help us improve our software,
you can enable telemetry by running the following command:

```
bash
nilup instrumentation enable --wallet <your-wallet-address>
```

The wallet address is optional.

While we will not collect any personal information, we still recommend using a new wallet address that cannot be linked to your identity by any third party.

By enabling telemetry you consent to the collection of telemetry data by the Nillion Network.
That includes but is not limited to
- The version of the SDK you are using
- The OS you are using
- The Processor Architecture you are using
- The SDK binary that you are running and the subcommand
- The wallet address you provided
- The errors produced by the SDK

For more information, our privacy policy is available at https://nillion.com/privacy/.

To disable telemetry run:

```
bash
nilup instrumentation disable
```

# showcase.md:

import ProjectsTable from '@site/src/components/ProjectsTable/index';

Awesome Projects

:::info
Some showcase examples use older Nillion SDK versions. For the latest code and approaches, refer to the Javascript + Python Clients.
:::

A showcase of open source projects built on Nillion by the community

<ProjectsTable/>

Add your project to the list

1. Create a new post in Github Discussions: Show and Tell.
2. Contribute to the nillion-docs repo with a pull request that adds your project information to awesome-projects.json.

# start-building.md:

Start Building

Start building on Nillion with one of our developer quickstart guides.

Write a Nada Program

The Nada Quickstart will teach you how to create a Nada project and write your first privacy-preserving Nada program. The Nada by Example Portal is an introduction to Nada with lots of example programs for your reference.

Build a Blind App

This Blind App Quickstart is ideal for developers interested in building frontends or fullstack web apps on Nillion. This quickstart will guide you through writing a Nada program, storing the program on the Nillion Netowrk, and creating a blind web app lets you store secrets and run your Nada program in the browser.

Connect a backend to Nillion

If you want to connect a backend to Nillion, check out the Nillion Python Quickstart.

Write a privacy-preserving Blind AI Nada program

To write privacy-preserving AI programs, check out our Nada AI examples, tutorials, and Google Colab links.

# store-secrets-js.md:

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';

Store Values

Store secret values (SecretBlob or SecretInteger values) in the network by the cluster, secret values, permissions, and payment receipt.

Create a SecretBlob to Store

Create a SecretBlob, with one or more secret strings encoded as bytearrays

```ts
const secretName = 'my-password'
const secretValue = 'abcdefg'
const nillionSecret = new nillion.NadaValues();
const byteArraySecret = new TextEncoder().encode(secretValue);
const secretBlob = nillion.NadaValue.newsecretblob(byteArraySecret);
nillionSecret.insert(secretName, secretBlob);
```

Create a SecretInteger to Store

Create a SecretInteger with one or more secrets values

```ts
const secretName = 'my-score'
const secretValue = 100
const nillionSecret = new nillion.NadaValues();
const secretInteger = nillion.NadaValue.newsecretinteger(
    secretValue.toString()
);
nillionSecret.insert(secretName, secretInteger);
```

Get a Quote to Store Values

<Tabs>

<TabItem value="getquote" label="Get quote to store secret" default>

```ts
const ttldays = 30; // set how long to store values
const storeOperation = nillion.Operation.storevalues(
    nillionSecret,
    ttldays
);

const quote = await getQuote({
    client: nillionClient,
    operation: storeOperation,
});
```

</TabItem>

<TabItem value="getQuote" label="getQuote helper">
ts reference showGithubLink
https://github.com/NillionNetwork/cra-nillion/blob/main/src/nillion/helpers/getQuote.ts

</TabItem>
</Tabs>

Pay to Store Values and get Payment Receipt

<Tabs>

<TabItem value="receipt" label="Payment receipt" default>
ts reference showGithubLink
https://github.com/NillionNetwork/cra-nillion/blob/main/src/nillion/components/StoreSecretForm.tsx#L133-L140

</TabItem>

<TabItem value="helpers" label="helper functions">
ts reference showGithubLink
https://github.com/NillionNetwork/cra-nillion/blob/main/src/nillion/helpers/nillion.ts#L24-L71

</TabItem>

</Tabs>

Store Secrets JavaScript Helper Function

ts reference showGithubLink
https://github.com/NillionNetwork/cra-nillion/blob/main/src/nillion/helpers/storeSecrets.ts

# store-secrets.md:

Store Secrets

Store secret values (SecretBlob or SecretInteger values) in the network by the cluster, secret values, permissions, and payment receipt.

:::tip

Open the Nillion Python Client Reference in another tab to search for available Nillion Client classes.

:::

Store a SecretBlob

Create a SecretBlob, with one or more secret strings encoded as bytearrays, get a quote to store values, pay to store values, get a payment receipt, and finally store the SecretBlob

python reference showGithubLink
https://github.com/NillionNetwork/python-examples/tree/main/examplesandtutorials/coreconceptstoreandretrievesecrets/storeandretrieveblob.py#L18-L66

Store a SecretInteger

Create a SecretInteger, with one or more secret integers, get a quote to store values, pay to store values, get a payment receipt, and finally store the SecretInteger

python reference showGithubLink
https://github.com/NillionNetwork/python-examples/tree/main/examplesandtutorials/coreconceptstoreandretrievesecrets/storeandretrieveinteger.py#L20-L66

# technical-reports-and-demos.md:

```
---
description: >-
  Technical reports, implementation examples, and browser-based demos on Nillion
  tech
---
```

Technical reports and demos

Technical reports

- Technical Report on Decentralized Multifactor Authentication
- Technical Report on Threshold ECDSA in the Preprocessing Setup
- More efficient comparison protocols for MPC

All technical reports are also available on Github

Implementation examples and tinydemos

Topic	Implementation Example	tinydemo
Nillion's secure multi-party computation (MPC) protocol	tinynmc library	-
Multifactor Authentication (MFA)	tinybio python library using tinynmc	tinybio: a secure decentralized biometric authentication demo
Signatures (Threshold ECDSA)	tinysig python library using tinynmc	-
Secure Auctions	tinybid python library using tinynmc	tinybid: a secure single-item first-price auction demo
3rd party API integration	tinybin demo code using tinynmc	tinybin: demo comparison of a 3p API output to a private input

# testnet-guides.md:

```
import DocCardList from '@theme/DocCardList';
```

Testnet Guides

Nillion has a four-phase strategy for deploying the Nillion Network.

During Phase 1: Genesis Sprint, it became possible to interact with the NilChain testnet (also known as the Coordination Layer) using NIL testnet tokens. Follow the guides below to create a wallet connected to the NilChain testnet, to use the Testnet Faucet, and to send NIL tokens on the NilChain testnet.

<DocCardList/>

We are now in Phase 2: From Genesis to Convergence, which means it is possible to interact with the Petnet testnet (also known as the Orchestration Layer) by storing data and performing blind computations over that data.

# welcome.md:

```
---
slug: /
---
```

import LinkButton from '@site/src/components/LinkButton/index';

Welcome to Nillion's Docs

Nillion is a secure computation network that decentralizes trust for high value data in the same way that blockchains decentralized transactions.

What if privacy was an enabler of use cases rather than a constraint?

Developers can use Nillion to build secure, high value data focused blind applications. Nillion empowers developers to

- write programs in Nada, our language for defining blind computation programs
- compile programs with the Nada compiler
- upload programs to the Nillion Network
- store permissioned secrets (high value data) on the Nillion Network
- run programs to perform blind computation on secrets on the Nillion Network

<br/>
<LinkButton text="Learn about Nillion" link="/what-is-nillion"/>

<LinkButton text="Build a Blind App"link="/quickstart"/>

# what-is-nillion.md:

What is Nillion

Nillion is a secure computation network that decentralizes trust for high value data in the same way that blockchains decentralized transactions.&#x20;

Traditional challenges of handling high value data include

- The need for secure storage: Traditionally, securing high value data requires encryption before storage. While this is effective for safeguarding the data at rest, it creates limitations when the data needs to be used or processed.
- The ability to compute on stored data without compromising security: Once data is encrypted and stored, the traditional process involves decrypting it to perform necessary computations, and then re-encrypting it. This decrypt-compute-reencrypt cycle not only creates potential security vulnerabilities, but also leads to inefficiencies in data handling.
- Achieving decentralization of data management: Traditionally when we think of

"decentralization", it's in the context of how blockchains have revolutionized trust assumptions for financial transactions. Applying this concept to high-value data management involves distributing data across multiple nodes, which is crucial for ensuring transparency, resilience, and independence from centralized control. However, this decentralization can introduce complexities in maintaining secure storage and efficient computation. The distributed nature of data can lead to challenges in achieving consistent security standards and efficient processing across the network, complicating data management.

Nillion addresses these challenges by leveraging privacy enhancing technologies (PETs) including Multi-Party Computation (MPC). These PETs enable users to securely store high value data on Nillion's peer-to-peer network of nodes, and allow for computations to be executed on the masked data itself. This eliminates the traditional need to decrypt data ahead of computation, enhancing the high value data's security.

High value data stored in the Nillion Network can be computed on while staying hidden, unlocking new use cases and verticals; early Nillion builders from our community are building things like tooling for private predictive AI and secure storage and compute solutions for healthcare, passwords, and trading data.

# addition.md:

Addition

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import TestProgram from '@site/src/components/TestProgram/index';

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/addition.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/additiontest.yaml

</TabItem>
</Tabs>

<TestProgram programName="addition"/>

# arg-max.md:

Arg Max

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import TestProgram from '@site/src/components/TestProgram/index';

Argmax is a data wrangling operation to find the index of the maximum value in a list of values.

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/argmax.py

```
</TabItem>

<TabItem value="test-1" label="Test 1">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
argmaxtest1.yaml

</TabItem>
<TabItem value="test-2" label="Test 2">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
argmaxtest2.yaml

</TabItem>
<TabItem value="test-3" label="Test 3">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
argmaxtest3.yaml

</TabItem>
<TabItem value="test-4" label="Test 4">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
argmaxtest4.yaml

</TabItem>
</Tabs>

<TestProgram programName="argmax" testFileName="argmaxtest1"/>


# cardio-risk.md:

Cardio Risk

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import TestProgram from '@site/src/components/TestProgram/index';

The cardio risk example program returns a risk score that depends on private
checks.

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/cardiorisk.py

</TabItem>

<TabItem value="test-0" label="Test 0">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
cardiorisk0test.yaml

</TabItem>

<TabItem value="test-1" label="Test 1">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
cardiorisk1test.yaml

</TabItem>
```

```
<TabItem value="test-2" label="Test 2">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
cardiorisk2test.yaml

</TabItem>

<TabItem value="test-3" label="Test 3">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
cardiorisk3test.yaml

</TabItem>
</Tabs>

<TestProgram programName="cardiorisk" testFileName="cardiorisk0test"/>

# comparison.md:

Comparison

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import TestProgram from '@site/src/components/TestProgram/index';

Less than <

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/comparisonlt.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
comparisonlttest.yaml

</TabItem>
</Tabs>

<TestProgram programName="comparisonlt"/>

Less than or equal to <=

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/comparisonlte.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
comparisonltetest.yaml

</TabItem>
</Tabs>
```

<TestProgram programName="comparisonlte"/>

Greater than >

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/comparisongt.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
comparisongttest.yaml

</TabItem>
</Tabs>

<TestProgram programName="comparisongt"/>

Greater than or equal to >=

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/comparisongte.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
comparisongtetest.yaml

</TabItem>
</Tabs>

<TestProgram programName="comparisongte"/>

# cube-root.md:

Cube Root

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import TestProgram from '@site/src/components/TestProgram/index';

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/cuberoot.py

</TabItem>

<TabItem value="test-1" label="Test 1">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/

cuberoottest1.yaml

</TabItem>
<TabItem value="test-2" label="Test 2">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
cuberoottest2.yaml

</TabItem>
<TabItem value="test-3" label="Test 3">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
cuberoottest3.yaml

</TabItem>
<TabItem value="test-4" label="Test 4">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
cuberoottest4.yaml

</TabItem>
</Tabs>

<TestProgram programName="cuberoot" testFileName="cuberoottest1"/>

# debugging.md:

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';

Debugging

When programming in Nada, keep this phrase in mind: Think Big, Code Small. Read about the "Think Big, Code Small" philosophy with Nada best practices and more debugging tips here.

Debugging with print()

When developing and debugging Nada programs, it can be helpful to inspect the values and types of variables at various points in your code. You can do this by adding Python print() statements, then running the file directly.

1. Include a Python main block

To use print statements for debugging in a Nada program, you can add the following Python main block to the end of your Nada program file. This block ensures that the nadamain() function runs standalone as the main program when the script is executed directly.

```
python
if name == "main":
   nadamain()
```

2. Add print() statements

Add print() statements within the Nada program to print variables or types of variables.

3. Run the program with Python

Run the program with Python to print any print() statements to the terminal.

```
python3 src/[programname].py
```

4. Remove print() statments before compiling a Nada program

Make sure to comment out or remove all print statements before compiling Nada programs, as Nada compilation will fail if any print statements are included.

The Python main block can be left in for Nada program compilation.

Debugging example

<Tabs>

<TabItem value="program" label="Nada program" default>

python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/debug.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/debugtest.yaml

</TabItem>
</Tabs>

Print type of variable

Printing the type of a variable prints the Nada data type to your terminal as a class. When debugging with type(), double check that your Nada operation supports the type printed by looking at the Nada Operations Supported Types column.

```
print(type(sum))
```

```
(.venv) ➜  nada-by-example git:(main) ✗ python3 src/debug.py
<class 'nadadsl.nadatypes.types.SecretInteger'>
```

Print variable

Printing a variable prints the Nada data type of the variable and any applicable operation types to your terminal. When debugging with operation types, check that the Nada operation type and syntax are correct by looking at the Nada Operations Syntax and Supported Types columns.

```
print(sum)
```

```
(.venv) ➜  nada-by-example git:(main) ✗ python3 src/debug.py
SecretInteger(inner=<nadadsl.operations.Addition object at 0x104ba5e20>)
```

Next steps

🐛 Now that you've learned to debug, you're ready to write and debug your own Nada programs in the src/ repo. Check out the Nada Operations page to see examples of supported Nada operations, their syntax, and supported data types.

# division.md:

Division

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import TestProgram from '@site/src/components/TestProgram/index';

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/division.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/divisiontest.yaml

</TabItem>
</Tabs>

<TestProgram programName="division"/>

# equality.md:

Equality (Equals)

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import TestProgram from '@site/src/components/TestProgram/index';

Private equality ==

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/equality.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/equalitytest.yaml

</TabItem>
</Tabs>

<TestProgram programName="equality"/>

Public output equality ==

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/
equalitypublic.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
equalitypublictest.yaml

</TabItem>
</Tabs>

<TestProgram programName="equalitypublic"/>

# first-program.md:

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';

Your First Nada Program

Welcome to your first Nada program example! This example is an addition program that is meant to help you understand the basics of writing a Nada program and introduce key concepts such as a Party, Input, SecretInteger, Operation, and Output.

This addition program in Nada demonstrates how to handle secret inputs from multiple parties, perform a computation (addition), and produce an output for another party.

Nada program and test file

<Tabs>

<TabItem value="program" label="Nada program" default>

The addition Nada program takes in a secret input from Alice and a secret input from Bob. It runs an addition operation to get the sum of the two secret inputs. The Nada program returns an output, the sum, to Charlie who never sees the values of Alice or Bob's secret inputs.

python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/addition.py

</TabItem>

<TabItem value="test" label="Test file">

You can auto-generate a test file like this one for any nada program with the nada tool. A test file verifies that the program works as expected. It provides test inputs and defines the expected outputs given the inputs. You'll learn how to run programs and test programs as part of the Nada by Example Local Setup Guide

yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
additiontest.yaml

</TabItem>
</Tabs>

Nada Program Key Concepts

Parties

A Party represents a user or entity participating in the computation. A party can provide inputs and/or receive outputs. The addition program has 3 parties named Alice, Bob, and Charlie. Alice and Bob provide inputs to the program. Charlie receives the output of the program.

python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/addition.py#L4-L6

Inputs

An Input is a value provided to a Nada program by a specific party. Inputs are wrapped with one of the following Public or Secret modes:

Public	Secret
PublicInteger	SecretInteger
PublicUnsignedInteger	SecretUnsignedInteger

The addition program has two SecretInteger typed Input:

1. num1 input by Alice
2. num2 input by Bob.

python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/addition.py#L7-L8

Operations

An operation is a computation performed on inputs to produce a result. In Nada, operations can be performed on both public and secret modes. Check out the full list of available Nada operations here.

The addition program involves one operation, addition, to sum the 2 secret inputs.

python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/addition.py#L9

Outputs

An Output defines how the result of the computation is shared with the parties. It ensures that the result of blind computation is properly revealed only to the intended party or parties.

python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/addition.py#L10

The addition program returns one output, the sum to Charlie. Charlie can see the resulting sum, but never sees the values of Alice or Bob's secret inputs that were added together to get the sum. This shows the power of blind computation!

Wrap up

In this example, you learned how to use Nada to create an addition program that performs blind computation. By defining Party, Input, SecretInteger, performing an Operation, and specifying Output, you can compute on secret inputs from multiple users. This ensures that sensitive data remains confidential throughout the computation process.

The power of Nada lies in its ability to compute on secret inputs without revealing the underlying data to any party involved. This makes Nada programs perfect for applications requiring high levels of privacy and security.

Start experimenting with Nada by learning how to run Nada by Example programs in the next section.

# for-loop.md:

For Loop

```
import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import TestProgram from '@site/src/components/TestProgram/index';

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/forloop.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
forlooptest.yaml

</TabItem>
</Tabs>

<TestProgram programName="forloop"/>
```

# helper-function.md:

Helper Functions

```
import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import TestProgram from '@site/src/components/TestProgram/index';

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/
listcomprehensionswithhelper.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
listcomprehensionswithhelpertest.yaml

</TabItem>
</Tabs>
```

<TestProgram programName="listcomprehensionswithhelper"/>

# if-else.md:

If / Else (Ternary)

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import TestProgram from '@site/src/components/TestProgram/index';

If / Else with a public condition

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/ifelsepublic.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
ifelsepublictest.yaml

</TabItem>
</Tabs>

<TestProgram programName="ifelsepublic"/>

If / Else with a private condition

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/ifelseprivate.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
ifelseprivatetest.yaml

</TabItem>
</Tabs>

<TestProgram programName="ifelseprivate"/>

# linear-scan.md:

Linear Scan

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import TestProgram from '@site/src/components/TestProgram/index';

The linear scan example scans a list of secret values from multiple parties, checking for the existence of specific values. The example uses a reusable helper function for the logic for determining whether a given value is present or isn't present in the list.

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/listscanlinear.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/listscanlineartest.yaml

</TabItem>
</Tabs>

<TestProgram programName="listscanlinear"/>

# list-comprehensions.md:

List Comprehensions

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import TestProgram from '@site/src/components/TestProgram/index';

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/listcomprehensions.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/listcomprehensionstest.yaml

</TabItem>
</Tabs>

<TestProgram programName="listcomprehensions"/>

# literal-data-type.md:

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import TestProgram from '@site/src/components/TestProgram/index';

Literals

Literals are constant values defined within the program rather than provided by a party.

Integer

Integer represents a literal integer value. This value can be a negative integer, a positive integer, or zero.

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/
additionliteral.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
additionliteraltest.yaml

</TabItem>
</Tabs>

<TestProgram programName="additionliteral"/>

UnsignedInteger

UnsignedInteger represents a literal unsigned integer value. This value can be zero or a positive integer.

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/
additionliteralunsigned.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
additionliteralunsignedtest.yaml

</TabItem>
</Tabs>

<TestProgram programName="additionliteralunsigned"/>

Boolean

Boolean represents a literal boolean value defined within the program rather than provided by a party. This value can be true or false.

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/
literalboolean.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
literalbooleantest.yaml

</TabItem>
</Tabs>

<TestProgram programName="literalboolean" />

```
# modulo.md:

Modulo

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import TestProgram from '@site/src/components/TestProgram/index';

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/modulo.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
modulotest.yaml

</TabItem>
</Tabs>

<TestProgram programName="modulo"/>

# multiplication.md:

Multiplication

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import TestProgram from '@site/src/components/TestProgram/index';

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/
multiplication.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
multiplicationtest.yaml

</TabItem>
</Tabs>

<TestProgram programName="multiplication"/>

# nada-ai.md:

import DocCardList from '@theme/DocCardList';

Nada AI

Nada-AI is a Python library designed for AI/ML on top of Nada DSL and Nillion
Network.

To learn more about the library, check out the full Nada-AI docs here.
```

:::info
Nada AI Limitations

The choice for blind computing implies certain trade-offs in comparison to conventional computing. What you gain in privacy, you pay in extra computational overhead & capacity constraints.

Therefore, you will notice that large-scale computational workloads may lead to long compilation and/or execution times or hitting network capacity guardrails. Each example lists model limitations or the number of input elements it has been tested with.

That said, the Nillion team is working around the clock to push the boundaries of this technology and bring the potential of blind computing to reality. 🚀
:::

Nada AI Examples with Google Colab Links

<DocCardList/>

# nada-data-types.md:

import NadaDataTypesTable from '../\data-types-table.mdx';

Primitive Modes and Data Types

This chart displays the primitive data types available in Nada, categorized by mode. The modes link to Nada program examples for each data type.

<NadaDataTypesTable/>

Secret and Public modes specify whether user inputs to a Nada program are treated as secret (private) or public data.

Literals are constants that can only be used within a Nada program and are not tied to specific user inputs.

# nada-numpy.md:

import DocCardList from '@theme/DocCardList';

Nada Numpy

Nada Numpy is a Python library designed for algebraic operations on NumPy-like array objects on top of Nada DSL and Nillion Network.

To learn more about the library, check out the full Nada-Numpy docs here.

Nada Numpy Examples with Google Colab Links

<DocCardList/>

# nada-operations.md:

import NadaOperationsTable from '../\operations-table.mdx';

Nada Operations

This chart shows all supported Nada operations, their syntax, and supported data types. Each operation has a linked example Nada program.

<NadaOperationsTable/>

# not.md:

Not

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import TestProgram from '@site/src/components/TestProgram/index';

The NOT operator, represented by the tilde symbol (), is used in the Nada DSL to invert or negate a boolean value. When applied to a boolean expression, the operator flips its value—turning True to False and False to True.

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/not.py

</TabItem>

<TabItem value="tie" label="Test 1">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/nottest1.yaml

</TabItem>
<TabItem value="rock" label="Test 2">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/nottest2.yaml

</TabItem>
</Tabs>

<TestProgram programName="not" testFileName="nottest1"/>

# power.md:

Power

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import TestProgram from '@site/src/components/TestProgram/index';

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/power.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/powertest.yaml

</TabItem>
</Tabs>

<TestProgram programName="power"/>

# probabilistic-truncation.md:

Probabilistic Truncation

import Tabs from '@theme/Tabs';

import TabItem from '@theme/TabItem';
import TestProgram from '@site/src/components/TestProgram/index';

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/
probabilistictruncation.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
probabilistictruncationtest.yaml

</TabItem>
</Tabs>

<TestProgram programName="probabilistictruncation"/>

# public-data-type.md:

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import TestProgram from '@site/src/components/TestProgram/index';

Public

The Public data type is used for inputs that are not sensitive and can be shown in the clear during computation. These input values are visible to the program.

PublicInteger

PublicInteger represents a user input public integer value. This value can be a negative integer, a positive integer, or zero.

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/
additionpublic.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
additionpublictest.yaml

</TabItem>
</Tabs>

<TestProgram programName="additionpublic"/>

PublicUnsignedInteger

PublicUnsignedInteger represents a user input public unsigned integer value. This value can be zero or a positive integer.

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/
additionpublicunsigned.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
additionpublicunsignedtest.yaml

</TabItem>
</Tabs>

<TestProgram programName="additionpublicunsigned"/>

PublicBoolean

PublicBoolean represents a user input public boolean value. This value can be true or false.

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/
publicconditional.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
publicconditionaltest.yaml

</TabItem>
</Tabs>

<TestProgram programName="publicconditional" />

# r-p-s.md:

Rock Paper Scissors

```
import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import TestProgram from '@site/src/components/TestProgram/index';
```

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/
rockpaperscissors.py

</TabItem>

<TabItem value="tie" label="Test - Tie">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
rockpaperscissorstie.yaml

</TabItem>

```
<TabItem value="rock" label="Test - Rock wins">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
rockpaperscissorsrock.yaml

</TabItem>
<TabItem value="paper" label="Test - Paper wins">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
rockpaperscissorspaper.yaml

</TabItem>
<TabItem value="scissors" label="Test - Scissors wins">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
rockpaperscissorsscissors.yaml

</TabItem>
</Tabs>

<TestProgram programName="rockpaperscissors"
testFileName="rockpaperscissorstie"/>

# random-number.md:

Random Number

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import TestProgram from '@site/src/components/TestProgram/index';

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/randomnumber.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
randomnumbertest.yaml

</TabItem>
</Tabs>

<TestProgram programName="randomnumber"/>

# reduce.md:

Reduce

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import TestProgram from '@site/src/components/TestProgram/index';

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/reduce.py

</TabItem>
```

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
reducetest.yaml

</TabItem>
</Tabs>

<TestProgram programName="reduce" />

# reveal.md:

Reveal

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import TestProgram from '@site/src/components/TestProgram/index';

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/reveal.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
revealtest.yaml

</TabItem>
</Tabs>

<TestProgram programName="reveal" />

# secret-data-type.md:

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import TestProgram from '@site/src/components/TestProgram/index';

Secret

The Secret data type is used for inputs that need to be kept confidential during computation. These input values are not visible to the program.

SecretInteger

SecretInteger represents a user input secret integer value. This value can be a negative integer, a positive integer, or zero.

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/
multiplication.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink

https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
multiplicationtest.yaml

</TabItem>
</Tabs>

<TestProgram programName="multiplication" />

SecretUnsignedInteger

SecretUnsignedInteger represents a user input secret unsigned integer value. This value can be zero or a positive integer.

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/
additionunsigned.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
additionunsignedtest.yaml

</TabItem>
</Tabs>

<TestProgram programName="additionunsigned" />

SecretBoolean

SecretBoolean represents a user input secret boolean value. This value can be true or false.

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/
secretconditional.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
secretconditionaltest.yaml

</TabItem>
</Tabs>

<TestProgram programName="secretconditional" />

# shift-left.md:

Left Shift

```
import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import TestProgram from '@site/src/components/TestProgram/index';
```

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/shiftleft.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
shiftlefttest.yaml

</TabItem>
</Tabs>

<TestProgram programName="shiftleft" />

# shift-right.md:

Right Shift

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import TestProgram from '@site/src/components/TestProgram/index';

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/shiftright.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
shiftrighttest.yaml

</TabItem>
</Tabs>

<TestProgram programName="shiftright" />

# shuffle.md:

Shuffle

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import TestProgram from '@site/src/components/TestProgram/index';

Simple Shuffle

This example uses the nada-numpy shuffle implementation to shuffle an array of four secret integers and return the shuffled values. This process preserves the original elements but places them in a different order.

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/shufflesimple.py

</TabItem>

<TabItem value="test" label="Test">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
shufflesimpletest.yaml

</TabItem>
<TabItem value="nadatest" label="nada-test">
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
shufflesimpletest.py

</TabItem>
</Tabs>

<TestProgram programName="shufflesimple" testFileName="shufflesimpletest"/>

More Shuffle Examples

This example demonstrates how the nada-numpy shuffling operation supports multiple data types, including Rational, SecretRational, PublicInteger, and SecretInteger. Shuffling can be applied using two approaches: the shuffle() function or the built-in .shuffle() method on arrays.

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/shuffle.py

</TabItem>

<TabItem value="test" label="Test">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
shuffletest.yaml

</TabItem>
</Tabs>

<TestProgram programName="shuffle" testFileName="shuffletest"/>

# square-root.md:

Square Root

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import TestProgram from '@site/src/components/TestProgram/index';

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/squareroot.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
squareroottest.yaml

</TabItem>
</Tabs>

<TestProgram programName="squareroot"/>

# standard-deviation.md:

Standard Deviation

```
import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import TestProgram from '@site/src/components/TestProgram/index';
```

Standard deviation is a measure of how spread out the values in a dataset are around the average. It gives you an idea of the typical distance between each data point and the mean.

```
<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/stdevinteger.py

</TabItem>

<TabItem value="test-1" label="Test 1">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
stdevintegertest1.yaml

</TabItem>
<TabItem value="test-2" label="Test 2">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
stdevintegertest2.yaml

</TabItem>
<TabItem value="test-3" label="Test 3">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
stdevintegertest3.yaml

</TabItem>
<TabItem value="test-4" label="Test 4">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
stdevintegertest4.yaml

</TabItem>
</Tabs>

<TestProgram programName="stdevinteger" testFileName="stdevintegertest1"/>
```

# subtraction.md:

Subtraction

```
import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import TestProgram from '@site/src/components/TestProgram/index';
```

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/subtraction.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
subtractiontest.yaml

</TabItem>
</Tabs>

<TestProgram programName="subtraction"/>

# variance.md:

Variance

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import TestProgram from '@site/src/components/TestProgram/index';

Variance is a measure of how spread out the values in a dataset are around the average. It shows how much the values differ from the mean.

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/
varianceinteger.py

</TabItem>

<TabItem value="test-1" label="Test 1">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
varianceintegertest1.yaml

</TabItem>
<TabItem value="test-2" label="Test 2">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
varianceintegertest2.yaml

</TabItem>
<TabItem value="test-3" label="Test 3">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
varianceintegertest3.yaml

</TabItem>
<TabItem value="test-4" label="Test 4">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
varianceintegertest4.yaml

</TabItem>
</Tabs>

<TestProgram programName="varianceinteger" testFileName="varianceintegertest1"/>

# voting.md:

Voting

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';
import TestProgram from '@site/src/components/TestProgram/index';

<Tabs>

<TabItem value="program" label="Nada program" default>
python reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/src/voting.py

</TabItem>

<TabItem value="test" label="Test file">
yaml reference showGithubLink
https://github.com/NillionNetwork/nada-by-example/blob/main/tests/
votingtest.yaml

</TabItem>
</Tabs>

<TestProgram programName="voting" />