# Nateve Documentation

***Release latest***

**Jul 29, 2022**

# CONTENTS

Nateve is a new general domain programming language open source inspired by languages like Python, C++, JavaScript, and Wolfram Mathematica.

Nateve is an transpiled language. Its first transpiler, Adam, is fully built using Python 3.8.

CHAPTER

ONE

# NATEVE PRINCIPAL FEATURES

## 1.1 1. Simple and easy to use

Just install the package and start coding.

## 1.2 2. Intuitive and easy to understand

Quickly understand the language and its features.

## 1.3 3. 100% free and open source

The language is free and open source. You can use it for any purpose. See the license.

## 1.4 4. 100% customizable

You can customize the language to your needs. You can make your own language from scratch. See the *Welcome to Nateve templates* section for more information.

CHAPTER

TWO

# WELCOME TO NATEVE TEMPLATES

Nateve Language includes a set of templates that can be used to customize Nateve. Templates are Python modules included in the templates subpackage. You can also create your own templates.

A template is a Python module that contains a set of words translations, functions definitions, and many other customizations. Every template can be used to customize Nateve. You just need to import the template with the `using` command and then use the template in the Nateve source code.

Learn more about templates in the *templates use* section.

CHAPTER

THREE

# WHY USE NATEVE TEMPLATES?

## 3.1 1. Customization

You can customize the language to your needs. Feel free to create your own templates or modify existing templates.

## 3.2 2. Team work

Your team can work together using different languages in the same file or project. For example, you can start coding in English and then switch to French.

It makes it easier to work together. Different team members can work on the same project using their native languages.

## 3.3 3. Easy to share

Your templates can be used by other developers. You can easily share your templates with the community.

CHAPTER

FOUR

# OPTIONS OF COMMAND LINE

1. `build`: Transpile Nateve source code to Python 3.8
2. `run`: Run Nateve source code
3. `transpile`: Transpile Nateve source code to an executable file (.exe)
4. `run-init-loop`: Run Nateve source code with an initial source and a loop source
5. *set-time-unit*: Set Adam time unit to seconds or miliseconds (default: milisecond)
6. `-v`: Activate verbose mode

CHAPTER

FIVE

# NATEVE TUTORIAL

In this tutorial, we will learn how to use Nateve step by step.

## 5.1 Step 0: Learn the basics

We recommend read the README.md file.

## 5.2 Step 1: Installation

Recommend Installation:

### 5.2.1 Clone the repo

```
$> git clone https://github.com/NateveLang/Adam.git
```

### 5.2.2 Add to path

### 5.2.3 Add your favorite templates

## 5.3 Step 2: Create a new Nateve file

```
$> cd my-project
$> COPY CON main.nate
```

## 5.4 Quick start examples

### 5.4.1 Hello World program

```
print("Hello, World!")
```

### 5.4.2 Is prime? program

```
def is_prime(n) {
    if n == 1 {
        return False
    }
    for i in range(2, n) {
        if n % i == 0 {
            return False
        }
    }
    return True
}

n = ninput("Enter a number: ")

if is_prime(n) {
    print("It is a prime number.")
}
else {
    print("It is not a prime number.")
}
```

## 5.5 Comments

If you want to comment your code, you can use:

```
~ This is a single line comment ~

~
    And this a multiline comment
~
```

## 5.6 Variables

This language uses variables. For declaring variables, you just need to write the name of the variable and the value of the variable.

For example:

```
a = 1                          ~ Interger ~
b = 1.0                        ~ Float ~
c = 1 + 2j                     ~ Complex ~
d = "hello"                    ~ String ~
e = True                       ~ Boolean ~
f = [1,2,3]                    ~ Vector ~
g = (1,2)                      ~ Tuple ~
h = Polynomial("1 +2x +x^2")   ~ Polynomial ~
i = $
```

(continues on next page)

(continued from previous page)

```
| 1 1 2 3 4 |
| 0 1 2 3 4 |
| 1 1 2 3 4 |
| 1 1 2 3 4 |
| 1 1 2 3 4 |
$                              ~ Matrix ~
```

Nateve allows data type as Integer, Float, Complex, Boolean, String, Tuple, None, Vector, Polynomial and Matrix.

## 5.7 Vectors

The Vectors allow to use all the data types before mentioned, as well as lists and functions.

Also, they allow to get an item through the next notation:

```
value_list = [1,2,3,4,5]
value_list2 = [0,1,0,1,0]

print(value_list[0])                ~ Output: 1 ~
print(value_list[0 : 4])            ~ Output: [1 2 3 4] ~

print(value_list.dot(value_list2))  ~ Output: 6 ~

print(value_list.add(value_list2))  ~ Output: [1 3 3 5 5] ~
```

## 5.8 Matrices

The Matrices are a special type of vectors of vectors.

```
a = $
| 1 5 |
| 0 2 |
$

b = $
|0 1|
|1 0|
$

print(a)
~ Output:
| 1 5 |
| 0 2 |
~

c = a.dot(b)
print(c)
~ Output:
| 5 1 |
```

(continues on next page)

(continued from previous page)

```
| 2 0 |
~

d = a.plus(b)
print(d)
~ Output:
| 1 6 |
| 1 2 |
~
```

## 5.9 Functions

For declaring a function, you have to use the next syntax:

```
def example_function(argument1, argument2, ...) {
    ~ sentence1 ~
    ~ sentence2 ~
    ...
    return Return_Value
}

example_function(argument1, argument2, ...) ~ Call the function ~
```

## 5.10 Conditionals

Regarding the conditionals, the syntax structure is:

```
if condition {
    ~ consequence ~
}
elif condition {
    ~ other_consequence ~
}
...
else {
    ~  default_consequence ~
}
```

For example:

```
if x <= 1 and x % 3 == 0 {
    a = 0
}
elif x == 9 {
    a = 1
}
else {
    a = 2
}
```

## 5.11 Loops

In order to use loops, you have to use the next syntax:

### 5.11.1 While Loop

```
while condition {
    ~ sentence1 ~
    ~ sentence2 ~
    ...
}
```

### 5.11.2 For Loop

```
for iterator in iterable {
    ~ sentence1 ~
    ~ sentence2 ~
    ...
}
```

CHAPTER

SIX

# USING TEMPLATES

One of the most important features of Nateve is the use of templates. Templates are a way to write code in a more readable way. They are words translations written in Python. In order to use templates, you just have to write the protected word "using", and then, write the name of the template. For example:

```
using "template_name"
```

Nateve includes the following standard templates:

1. "english": This template is used to write the code of the program in English. It is the default template.
2. "spanish": This template is used to write the code of the program in Spanish.
3. "french": This template is used to write the code of the program in French.

You also can use your own templates. Just create a file with the name of the template and write the code of the template in the file. Here is a blank template:

```
# The name of the transpiler. This line is required. Do not change it.
transpiler_name = "adam"

"""
The following code is the translation of the code.
You can write your code here and modify the content of the variables.
Do not change the name of the variables.
"""

# All the symbols that the transpiler uses.
mayusc = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
alphabet = mayusc + mayusc.lower() + "_"
digits = "0123456789"
alphanum = alphabet + digits
blanks = "/t /n"
strings = ["'", '"', '"""', "'''"]
matrices = "$"
vectors = "[]"
embedded = "°"
commentaries = "~"
floating = "."
one_char_symbols = "+-*/%=<>()[]{}#@,."
two_char_symbols = ["//", "==", "<=", ">="]

# All the data types that the transpiler uses.
FLOAT = "float"
```

(continues on next page)

(continued from previous page)

```
INT = "int"
COMPLEX = "complex"
STRING = "string"
DOCSTRING = "docstring"
NULL = "none"
MATRIX = "matrix"
VECTOR = "vector"

# All the keywords that the transpiler uses.
USE, INCLUDE = "using", "include"
IMPORT, FROM, AS, PASS, IN = "import", "from", "as", "pass", "in"
IF, ELIF, ELSE = "if", "elif", "else"
TRY, EXCEPT, WITH = "try", "except", "with"
WHILE, FOR, BREAK, CONTINUE = "while", "for", "break", "continue"
OPERATOR, RETURN = "def", "return"
CLASS, SELF = "class", "self"
AND, OR, NOT, TRUE, FALSE = "and", "or", "not", "True", "False"

# All the status codes that the transpiler uses.
embedding = 200
identifier = 300
eof = 400

# All extra functions that the transpiler uses. Feel free to add your own functions.
# The string special_functions is used to write these functions.
# You can use variables in it using the fstring notation.
special_functions = f"""
def ninput(prompt = '', default = ''):
    return float(input(prompt, default))

def binput(prompt = '', default = ''):
    return bool(input(prompt, default))

def update_std():
    subprocess.call([sys.executable, '-m', 'pip', 'install', 'eggdriver'])
"""
```

CHAPTER

SEVEN

# SOME EXAMPLES

## 7.1 Example 1

```
~Nateve Example 1~

update_std()  ~update std library~

for i in range(2) {
    print(i)
}

install("matplotlib")

try {
    print(2/0)
}

except {
    print("xd")
}
```

Output:

```
0
1
matplotlib successfully installed
xd
```

## 7.2 Example 2

```
~Nateve Example 2~

theta = pi/3
print(sin(theta), cos(theta), tan(theta))

p = sin_serie
print(p.eval(theta))
```

(continues on next page)

(continued from previous page)

```
derive(p)

print(p.eval(theta))

import numpy as np
x = "hello"
c = Matrix("""
| 1 1 2 3 4 |
| 0 1 2 3 4 |
| 1 1 2 3 4 |
| 1 1 2 3 4 |
| 1 1 2 3 4 |
""")
c.display()

a = Vector("[ 1 2 3 4 5 6 30 0 9]")
a.display()
```

Output:

```
0.8660254037844386 0.5000000000000001 1.73205080756887
0.8660254037844386
0.5000000000000001
| 1 1 2 3 4 |
| 0 1 2 3 4 |
| 1 1 2 3 4 |
| 1 1 2 3 4 |
| 1 1 2 3 4 |
[ 1 2 3 4 5 6 30 0 9 ]
```

## 7.3 Example 3

```
~Nateve Example 3~

using "spanish"

theta = pi/3
imprime(sen(theta), cos(theta), tan(theta))

p = serie_sen
imprime(p.eval(theta))

deriva(p)

imprime(p.eval(theta))

importa numpy como np
x = "hello"
c = Matriz("""
| 1 1 2 3 4 |
```

(continues on next page)

(continued from previous page)

```
| 0 1 2 3 4 |
| 1 1 2 3 4 |
| 1 1 2 3 4 |
| 1 1 2 3 4 |
""")
c.display()

a = Vector("[ 1 2 3 4 5 6 30 0 9]")
a.display()
```

Output:

```
0.8660254037844386 0.5000000000000001 1.73205080756887
0.8660254037844386
0.5000000000000001
| 1 1 2 3 4 |
| 0 1 2 3 4 |
| 1 1 2 3 4 |
| 1 1 2 3 4 |
| 1 1 2 3 4 |
[ 1 2 3 4 5 6 30 0 9 ]
```

## 7.4 Example 4

```
~Nateve Example 4~

using "spanish"

amo_Nateve = verdadero

si  amo_Nateve == verdadero {
    imprime("Yo amo Nateve!")
}

delocontrario {
    imprime("Odio Nateve :c")
}

usando "english"

if 1 < 3 {
    print("Try Nateve!")
}
else {
    print("NO")
}

using "french"

v = "Bonjour"
```

(continues on next page)

(continued from previous page)

```
imprimer(v, "Nateve!")
```

Output:

```
Yo amo Nateve!
Try Nateve!
Bonjour Nateve!
```

## 7.5 Example 5

```
~Nateve Example 5~

include "example4.nate"

using "spanish"

imprime("Nateve example 5")
```

Output:

```
Yo amo Nateve!
Try Nateve!
Bonjour Nateve!
Nateve example 5
```

## 7.6 Example 6

```
~Nateve Example 6~

using "spanish"

incluye "example5.nate"

a = $
| 1 5 |
| 0 2 |
$

b = $
|0 1|
|1 0|
$

imprime("a = ")
imprime(a)

imprime("b = ")
imprime(b)
```

(continues on next page)

(continued from previous page)

```
c = a.dot(b)

imprime("a * b =")
imprime(c)

imprime("a + b =")
print(a.plus(b))

d = [1, 2, 3, 4, 5]
imprime(d)

e = [0, 1, 0, 1, 0]
imprime(e)

f = d.dot(e)
imprime(f)

g = d.plus(e)
imprime(g)

~ using spanish, "y" means "and".
Then, we need to use other template like french ~

using "french"

definir r(x, y, z){
retourner $
|x|
|y|
|z|
$
}

x, y, z = 1, 5, 3

j = r(x, y, z)

imprimer(j)

k = $
|2 0 0|
|0 2 0|
|0 0 2|
$

imprimer(k.dot(j))
```

Output:

```
Yo amo Nateve!
Try Nateve!
```

(continues on next page)

(continued from previous page)

```
Bonjour Nateve!
Nateve example 5
a =
| 1 5 |
| 0 2 |

b =
| 0 1 |
| 1 0 |

a * b =
| 5 1 |
| 2 0 |

a + b =
| 1 6 |

[1, 2, 3, 4, 5]
[0, 1, 0, 1, 0]
6
[1, 3, 3, 5, 5]
| 1 |
| 5 |
| 3 |

| 2 |
| 10 |
| 6 |
```

CHAPTER

EIGHT

# FEEDBACK

I would really appreciatte your feedback. You can submit a new issue.

CHAPTER

NINE

# CONTRIBUTE

This is an **opensource** project, everyone can contribute and become a member of the community of **Nateve**.

CHAPTER

TEN

# WHY BE A MEMBER OF THE NATEVE COMMUNITY?

## 10.1 1. A simple and understandable code

The source code of Adam is made with Python 3.8, a language easy to learn, also good practices are a priority for this project.

## 10.2 2. A great potencial

This project has a great potential to be the next programming language for education, to develop the quantum computing, and to develop the AI.

## 10.3 3. Simple

One of the main purposes of this programming language is to create an easy-to-learn language, which at the same time is capable of being used for many different purposes.

## 10.4 4. Respect for diversity

Everybody is welcome, it does not matter your genre, experience or nationality. Anyone with enthusiasm can be part of this project. Anyone from the most expert to the that is beginning to learn about programming, marketing, design, or any career.

CHAPTER

# ELEVEN

# HOW TO START CONTRIBUTING?

There are multiply ways to contribute, since sharing this project, improving the brand of SigmaF, helping to solve the bugs or developing new features and making improves to the source code.

- **Share this project**: You can put your star in the repository, use the topic nateve or talk about this project. You can use the hashtag #Nateve in Twitter, LinkedIn or any social network too.
- **Improve the brand of Nateve**: If you are a marketer, designer or writer, and you want to help, you are welcome.
- **Help to solve the bugs**: if you find one bug notify us an issue. On this we can all improve this language.
- **Developing new features**: If you want to develop new features or making improvements to the project, you can do a fork to the `dev` branch (here are the ultimate develops) working there, and later do a ``*pull request*` <https://docs.github.com/en/github/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/creating-a-pull-request>`_ to `dev` branch in order to update **Nateve**.

CHAPTER

# TWELVE

# NQS INSIDE

## 12.1 NQS: Natural Quantum Script. A special domain programming language that aims to simplify the first contact with quantum computing

Natural Quantum Script is a special domain programming language that aims to simplify the first contact with quantum computing for people who have prior knowledge in quantum circuits, but not in quantum software development.

Scripts written in NQS seek to visually resemble quantum circuits as much as possible. For example:

```
q0 q1
X
H
.--- X
c1
```

NQS is based on Qiskit, but seeks to go mainstream in the future. This is an OS project whose initial goal was to make it easier to write basic scripts in Qiskit and to bridge the gap for people who don't dare to delve into quantum computing.

CHAPTER

# THIRTEEN

# EGGDRIVER STANDARD LIBRARY

Most of Nateve functions, variables and classes are implemented in the **Eggdriver Standard Library**.

In this section we will see how to use the Eggdriver Standard Library features.

## 13.1 1. ceil

Return the ceil function to a number.

```
result = ceil(pi)
print(result)
```

```
4
```

## 13.2 2. clearConsole

Clear the console.

```
clearConsole() ~ clear the console ~
```

## 13.3 3. cos

Return the cosine function of a number.

```
result = cos(pi/2)
print(result)
```

```
0
```

## 13.4 4. display

Display a text in the console each certain number of milliseconds, while a condition is true. The default condition is `true`.

```
display("Hello world!", 1000, 1 > 0) ~ display the text "Hello world!" for 1 second ~
```

Each 1 second:

```
Hello world!
```

## 13.5 5. e

The Euler number with 73 digits of precision.

e = 2.7182818284590452353602874713526624977572470936999595749669676277240766303

## 13.6 6. floor

Return the floor function to a number.

```
result = floor(pi)
print(result)
```

```
3
```

## 13.7 7. get

Get an input, with a tag.

```
input_string = get("my-console-application")
print(input_string)
```

Input/Output:

```
$my-console-application> Hi
Hi
```

## 13.8 8. inf

The computable infinity used for limits calculation.

inf = 10 ** 11

## 13.9 9. itself

The identity function.

```
result = itself(10)
print(result)
```

```
10
```

## 13.10 10. ln

Return the natural logarithm function of a number.

```
result = ln(1)
print(result)
```

```
0
```

## 13.11 11. log

Return the logarithm function of a number, with a certain base. You can set an ansatz domain in order to improve the speed of the computation, using the `domain` parameter.

```
result = log(e ** 2, e, domain = [1, 4])
print(result)
```

```
2
```

## 13.12 12. pg

Print content in white and get an input with a tag. The default tag is 'egg'.

```
input_string = pg('Hello world!', "my-console-application")
print(input_string)
```

Input/Output:

```
Hello world!
$my-console-application> Hi
Hi
```

## 13.13 13. pi

The number pi with 73 digits of precision.

pi = 3.14159265358979323846264338327950288419716939937510582097494459230781640 62

## 13.14 14. Polynomial

The Polynomials class. It allows to use Polynomials and Series.

You can create a Polynomial using the **vector syntax**:

```
p = Polynomial([1, 2, 3])
```

or using the **literal syntax**:

```
p = Polynomial("1 +2x +3x^2")
```

Using the literal syntax, **you can add literal polynomials in the initialisation**:

```
p1 = Polynomial("1 +2x +3x^2" + "-4x")
p2 = Polynomial("1 +2x +3x^2 -4x")
p1.display()
p2.display()
```

```
1 -2x +3x^2
1 -2x +3x^2
```

Example of use:

```
result = Polynomial("1 2x +x^2")
print(result)

result_list = list(result)
print(result_list)
```

```
1 2x +x^2
[1, 2, 1]
```

### 13.14.1 Polynomial.degree

Return the degree of a Polynomial.

```
poly = Polynomial("1 +3x^4 +x^3")
deg = poly.degree
print(deg)
```

```
4
```

### 13.14.2 Polynomial.display

Display a Polynomial in the console.

```
poly = Polynomial([1, 0, 3, 1])
poly.display()
```

```
1 +3x^2 +x^3
```

### 13.14.3 Polynomial.eval

Return the evaluation of a Polynomial at a certain point.

```
poly = Polynomial("1 -x^2")
result = poly.eval(7)
print(result)
```

```
-48
```

### 13.14.4 Polynomial.power

Return the power of a Polynomial to a certain power.

```
poly = Polynomial("1 +x")
result = poly.power(2)
result.display()
```

```
1 +2x +x^2
```

### 13.14.5 Polynomial.plus

Return the sum of two Polynomials.

```
poly1 = Polynomial("1 +3x^2 +x^3")
poly2 = Polynomial("4x +6x^3")
poly = poly1.plus(poly2)
poly.display()
```

```
1 +4x +3x^2 +7x^3
```

### 13.14.6 Polynomial.times

Return the product of two Polynomials.

```
poly1 = Polynomial("1 - 3x^2")
poly2 = Polynomial("5x^3")
poly = poly1.times(poly2)
poly.display()
```

```
5x^3 -15x^5
```

## 13.15 15. Polynomial.var

The variable of a Polynomial.

```
poly = Polynomial("1 - 3x^2")
v = poly.var
print(v)
```

```
x
```

### 13.15.1 Polynomial.zeros

Return the zeros of a Polynomial.

```
poly = Polynomial("1 - x^2")
z = poly.zeros
print(z)
```

```
[-1, 1]
```

## 13.16 16. pow

Pow a number to a certain power.

```
result = pow(2, 3)
print(result)
```

```
8
```

## 13.17 17. ProgressBar

A progress bar pip-like for console implementations.

```
bar = ProgressBar()
bar.iterate(printPercent = True)
```

Last iteration:

```
||      100%
```

### 13.17.1 ProgressBar.bar

Returns a ProgressBar as a text, with a certain length and percent of progress.

```
p_bar = ProgressBar()
text = p_bar.bar(0.5, 16)
print(text)
```

```
|        |      50%
```

### 13.17.2 ProgressBar.display

Display a progress bar in the console, with a certain length and percent of progress, waiting a certain number of milliseconds. You can also set the `printPercent` parameter to `True` to print the percent of progress.

```
p_bar = ProgressBar()
p_bar.display(0.75, 16, 1000, printPercent = False)
```

```
|    |
```

### 13.17.3 ProgressBar.iterate

For each percent of progress, display a progress bar in the console, with length 32 and a sleeping time of 24 milliseconds. You can choose a function to execute at each iteration.

```
p_bar = ProgressBar()

def my_function() {
    print("Hello world!")
    clearConsole()
}

p_bar.iterate(my_function, printPercent = True)
```

Iteration 25:

```
Hello world!
|                         |      25%
```

Last iteration:

```
Hello world!
||      100%
```

## 13.18 18. put

Print content in a certain eggdriver color. The default color is white. You can set the color to “” to reset the color. You can also set an ending string using the `end` parameter.

```
put("Hi", "", ";")
```

```
Hi;
```

## 13.19 19. R

The Reals numbers set.

R = [-inf, inf]

## 13.20 20. series_inf

The computable infinity used for series calculation.

series_inf = 500

## 13.21 21. sin

Return the sine function of a number.

```
result = sin(pi/2)
print(result)
```

```
1
```

## 13.22 22. sleep

Wait a certain number of milliseconds.

```
sleep(1000) ~ sleep for 1 second ~
```

## 13.23 23. sysCommand

Execute a python command. (Currently only for Windows).

```
sysCommand("-m pip install --upgrade pip") ~ execute the command "py -m pip install --
↪upgrade pip" ~
```

## 13.24 24. tan

Return the tangent function of a number.

```
result = tan(pi/4)
print(result)
```

```
1
```

## 13.25 25. truncate

Truncate a number to a certain number of digits.

```
result = truncate(pi, 5)
print(result)
```

```
3.14165
```

## 13.26 26. x

The x variable. Used for Polynomials and Series evaluation.

```
result = x(2)
print(result)
```

```
2
```

CHAPTER

# FOURTEEN

# TEMPLATES STANDARD LIBRARY