Typethon is a versatile programming language that is derived from Python and empowers developers with the added benefits of static typing, referencing, and modern language features. Being statically and strongly typed, Typethon ensures that code is more reliable and less prone to errors. With its ease of use and flexibility, it allows developers to write clean and concise code. This documentation provides an in-depth overview of Typethon's syntax, features, and usage, making it easier for developers to understand and leverage the power of this language.

Getting Started

To start programming in Typethon, you must install the Typethon compiler on your system. Once installed, you will be able to effortlessly create source code files with a .typ extension, and subsequently compile and run them with ease.

Syntax Overview

Typethon is a programming language that closely resembles Python, but it comes with its own set of constructs and keywords. To give you a brief idea of Typethon's syntax, here are a few key points to keep in mind:

- Blocks of code are defined using curly braces {} for static scoping.
  - Variables and constants are declared explicitly with their types.
- Pointers can be used for referencing memory addresses.

Start with a simple "Hello World" program to familiarize yourself with Typethon syntax:

```
def main() : none {
    print("Hello, Typethon!")
}
```

Simple Data

In Typethon, data types categorize the kinds of values that variables and expressions can hold or evaluate to. Understanding data types is crucial for writing robust and efficient code. Typethon supports various data types, including primitive types and collection types.

Primitive Types

Primitive types are fundamental data types that represent singular values. They are immutable, which means that their values cannot be altered once they are created. Here is a list of primitive types supported in Typethon:

int

The int type represents integers, which are whole numbers without any fractional part. In Typethon, integers can be positive, negative, or zero.

```
age: int = 30
count: int = -5
```

float

The float type represent floating-point numbers, which include both integers and numbers with a fractional part.

```
pi: float = 3.14159
height: float = 5.8
```

str

The str type represents strings, which are sequences of characters enclosed within single or double quotes.

```
name: str = "Typethon"
message: str = 'Hello, World!'
```

chr

The chr type represents a single character.

```
first_initial: chr = 'T'
```

bool

The bool type represents boolean values, which can be either True or False.

```
is_typethon_fun: bool = True
is_typethon_complex: bool = False
```

none

The none type represents a special value indicating the absence of a value.

```
result: none = None
```

Collection Types

List

```
numbers: list = [1, 2, 3, 4, 5]
```

Dictionary

```
person: dict = {'name': 'John', 'age': 30, 'city': 'New York'}
```

Set

```
unique_numbers: set = {1, 2, 3, 4, 5}
```

Tuple

```
coordinates: tuple = (10, 20)
```

Variable Declaration

Variable declaration in Typethon involves assigning a data type to a variable. It follows the
syntax

```
var <identifier> : <type>

count: int = 5

name: str = "Typethon"
```

Pointer Declaration

Pointer declaration in Typethon involves creating a pointer variable that stores the memory address of another variable. It follows the syntax

```
ptr <identifier> : <type>
ptr_var: ptr int
```

Class Declaration

Class declaration in Typethon defines a new class. It follows the syntax.

```
class <identifier>.
class MyClass
```

Object Declaration

Object declaration in Typethon involves creating an instance of a class. It follows the syntax

```
var <identifier> = <Class>()
obj: MyClass = MyClass()
```

Assignment

The assignment statement in Typethon assigns a value to a variable. It follows the syntax

```
<identifier> = <expression>
x = 10
name = "Typethon"
```

If-Else statements

The if statement in Typethon is used for conditional branching. It can include if, else, and elif clauses.

```
if x > 5:{

    print("x is greater than 5")
    }
else:{
    print("x is not greater than 5")
}
```

While Statement

While a statement in Typethon is used to repeatedly execute a block of code as long as a condition is true.

```
while x < 10:{
    print(x)
    x += 1
}
```

For Statement

For statement in Typethon is used to iterate over a sequence (such as a list or tuple) or a range of values.

```
for i in range(5):{
    print(i)
}
```

Function Definition

Function definition in Typethon declares a new function. It follows the syntax

```
def <identifier>() : <type> { <statements> }.

def greet(): none {
    print("Hello, Typethon!")
}
```

Return Statement

Return statement in Typethon is used to return a value from a function.

```
def add(x, y): int {
    return x + y
}
```

Continue Statement

The continue statement in Typethon is used to skip the remaining code inside a loop and continue with the next iteration.

```
for i in range(10):{
    if i % 2 == 0:{
        continue
    print(i)
}
}
```

Break Statement

A break statement in Typethon is used to exit the loop prematurely.

```
while True:{
    x += 1

    if x == 10:{
        break
}
}
```

Arithmetic Expressions

Arithmetic expressions in Typethon are used to perform mathematical operations such as **addition, subtraction, multiplication, division, and modulus.**

```
result = 10 + 5
difference = 10 - 5
product = 10 * 5
quotient = 10 / 5
remainder = 10 % 3
```

Comparison Expressions

Comparison expressions in Typethon are used to compare values and evaluate a boolean result.

```
is_equal  (x == y)

not_equal  (x != y)

greater_than  (x > y)

less_than_or_equal  (x <= y)
```

Boolean Expressions

Boolean expressions in Typethon are used to perform logical operations such as AND, OR, and NOT.

```
result = (x > 5) and (y < 10)

result = (x < 5) or (y > 10)

result = not (x == y)
```

Function Calls

Function calls in Typethon are used to invoke functions with or without arguments.

```
result = add(5, 10)

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

Literals

Literals in Typethon represent fixed values such as integers, floats, strings, and boolean values.

integer_literal = 10

float_literal = 3.14

string_literal = "Typethon"

boolean_literal = True

Collections

Collections in Typethon represent groups of items such as lists, dictionaries, sets, and tuples.

```
list_example = [1, 2, 3, 4, 5]

dict_example = {'a': 1, 'b': 2, 'c': 3}

set_example = {1, 2, 3, 4, 5}

tuple_example = (1, 2, 3, 4, 5)
```

Range Expressions

Range expressions in Typethon are used to generate sequences of numbers within a specified range.

```
for i in 1..10:{
    print(i)
}
```

```
>> 1, 2, 3, 4, 5, 6, 7, 8, 9
```

Example (inclusive range):

```
for i in 1..=10:{

    print(i)
}
```

```
>> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

Lambda Expressions

Lambda expressions in Typethon are used to create anonymous functions.

```
square = lambda x: x * x
```

Syntax Documentation

Main Function Syntax

```
<MainFunction> ::= "def" "main" "(" ")" ":" "none" "{" <Program>
"}"
```

```
def: Keyword indicating the definition of a function.

main: Name of the main function.

(): Parentheses indicating the absence of parameters for the main
function.

:: Colon marking the beginning of the function body.
```

```
none: Return type of the main function, indicating no return value.

{}: Curly braces enclosing the body of the main function.

<Program>: Placeholder for the program statements within the main function.
```

```
def main() : none {
    x = 5
    y = 10
    result = x + y
    print("The result is:", result)

}
```

Program Syntax

```
<Program> ::= <StatementList>
```

- StatementList: Placeholder for a list of statements that constitute the program.
- The Program syntax defines the structure of a program, specifying that it consists of a list of statements

```
def calculate_sum() : none {
    for i in range(5):
        print("Iteration:", i)
}
```

Statement List Syntax

The IfStatement includes an if-else block. If the condition x > 5 evaluates to true, the first print statement will be executed; otherwise, the second print

statement will be executed.

```
x = 3
if x > 5 {
    print("x is greater than 5")
} else {
    print("x is not greater than 5")
}
```

Comments Syntax

The Comments syntax allows you to include explanatory notes and documentation within their Typethon code. Single-line comments begin with the # symbol and extend until the end of the line. Multi-line comments are enclosed within triple quotes (""") or triple single quotes (''), allowing for comments spanning multiple lines.

Single Line Comments

```
# This is a single-line comment

x = 5  # Assigning a value to variable x
```

Multi-Line Comments

```
"""
This is a multi-line comment.
It spans multiple lines.
"""
x = 10
```

Constant Declaration Syntax

The ConstantDeclaration syntax allows to declare constants with fixed. Constants are assigned values during declaration, and once initialized, their values cannot be changed.

```
const PI: float = 3.14159

const MAX_VALUE: int = 100

const DEFAULT_PERSON = Person("John", 30)
```

Pointer Declaration Syntax

The PointerDeclaration syntax allows to declare pointer variables. Pointers store memory addresses of other variables or objects, allowing indirect access to their data.

Simple Pointer Declaration

```
ptr x: int

Pointer Declaration for Object

ptr obj_ptr: MyClass
```

Return Statement Syntax

Allows functions to return control to the calling code and optionally pass back a value. The returned value, if provided, can be used by the caller for further computation or processing.

Return Statement without Value

```
def calculate_sum(a, b): int {

```

```
    return a + b
}
```

Function Application Syntax

The FunctionApplication syntax in Typethon allows programmers to invoke functions by specifying their names and passing arguments within parentheses.

Simple Function Call

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

In this example, the print function is called to display the string "Hello, Typethon!" to the console. No arguments are passed to the function in this case.

Function Call with Arguments

```
result = add_numbers(5, 10)
```

The add_numbers function is called with two arguments 5 and 10. The function computes the sum of the two numbers and returns the result.

Function Definition Syntax

The FunctionDefinition allows programmers to define functions with specified parameters, return types, and a sequence of statements that define the functionality of the function.

Simple Function Definition

```
def greet(name: str) -> none {

    print("Hello, " + name + "!")
```

```
}
```

Function Definition with Return Value

```
def add(a: int, b: int) -> int {

    return a + b

}
```

Parameter List Syntax

The ParameterList allows to specify the parameters accepted by a function.

Single Parameter

```
def greet(name: str) -> none {

    print("Hello, " + name + "!")

}
```

Multiple Parameters

```
def add(a: int, b: int) -> int {

    return a + b

}
```

Range Expression Syntax

The ReferenceExpression allows us to obtain the memory address of a variable. Prefixing an identifier with the ampersand operator (&), programmers can retrieve the memory address where the variable is stored in memory.

Reference Expression

```
x = 10

address = &x
```

The RangeExpression enables the creation of ranges with the specified start and end values. Ranges can be inclusive or exclusive.

Range Expression

```
inclusive_range = 1..=5

exclusive_range = 1..10
```