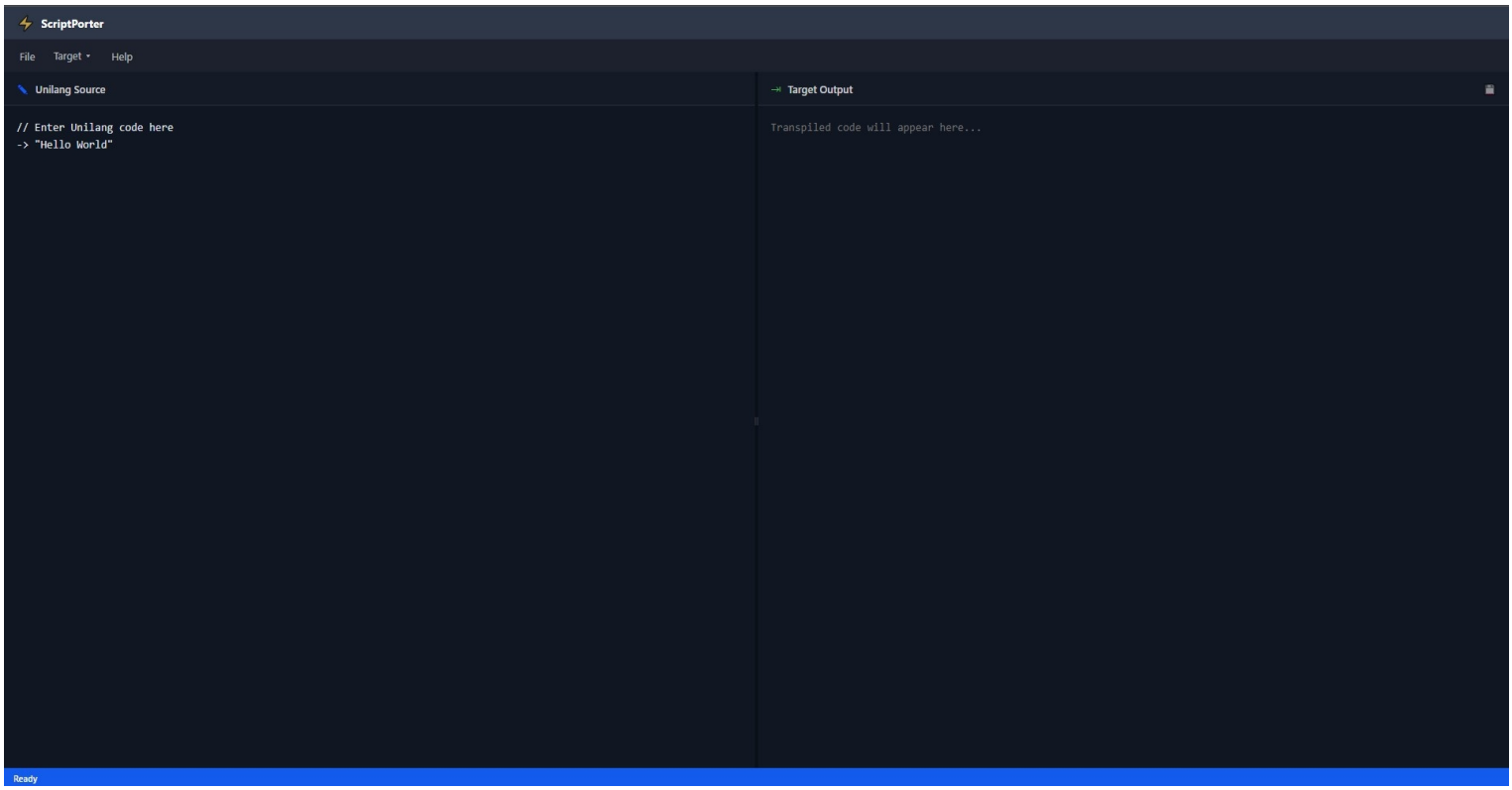
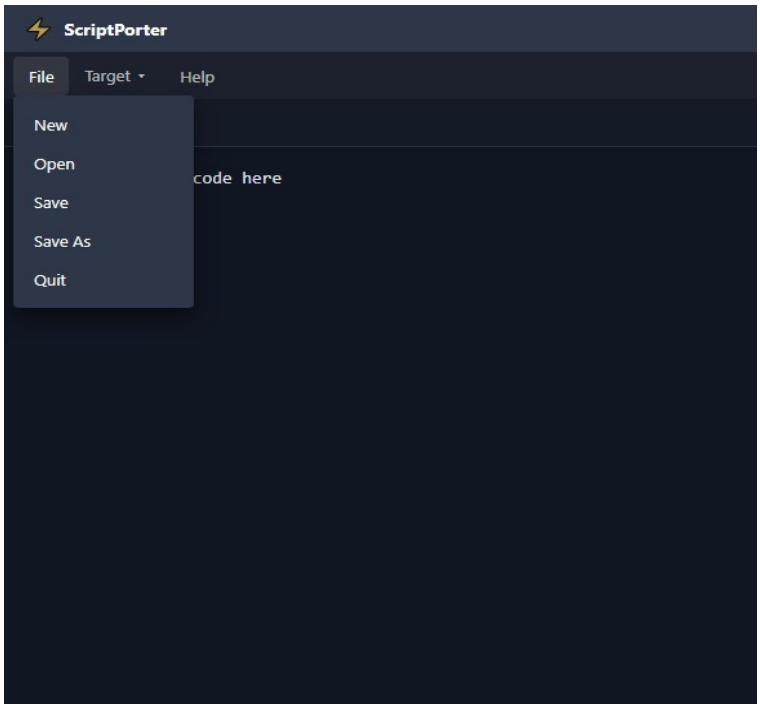

Script
Porter

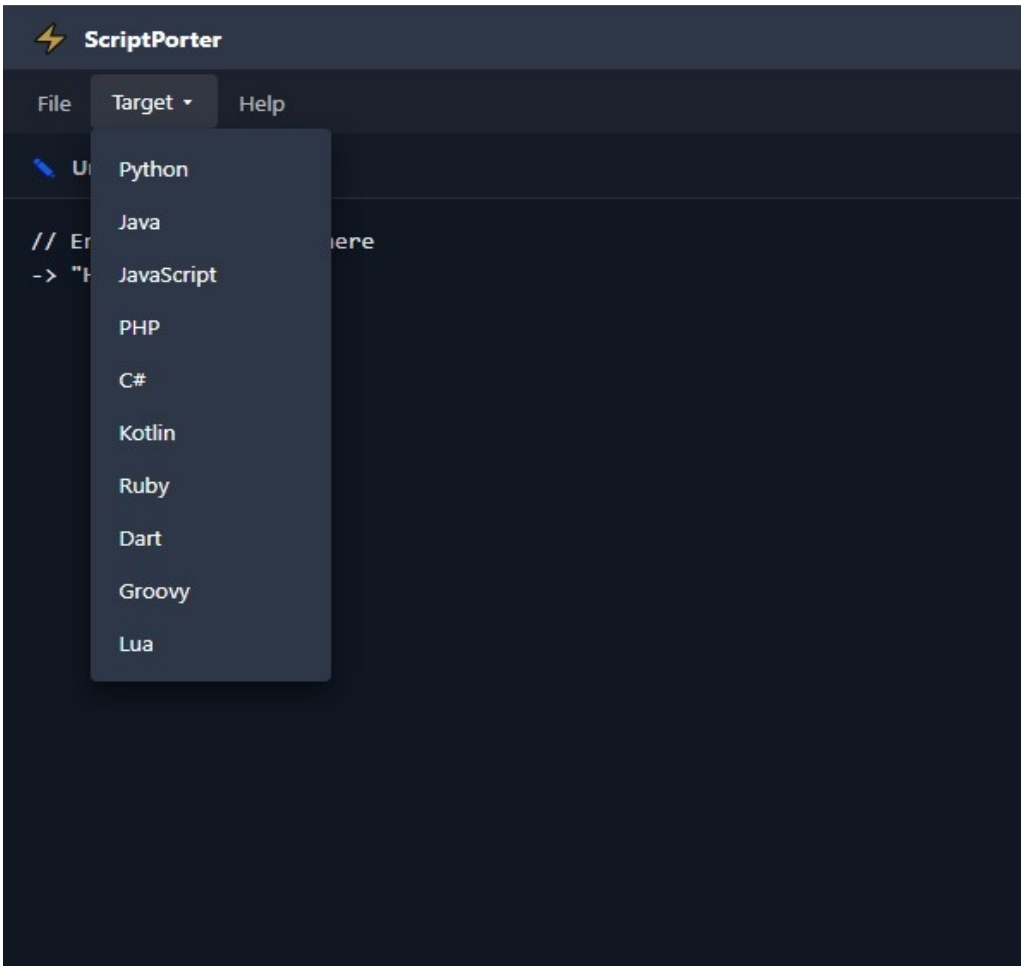
Main window(opening window)



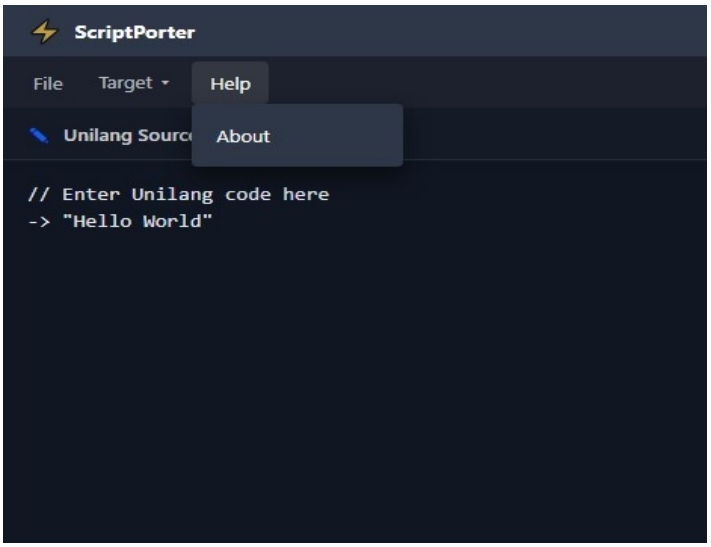
File menu



Target menu

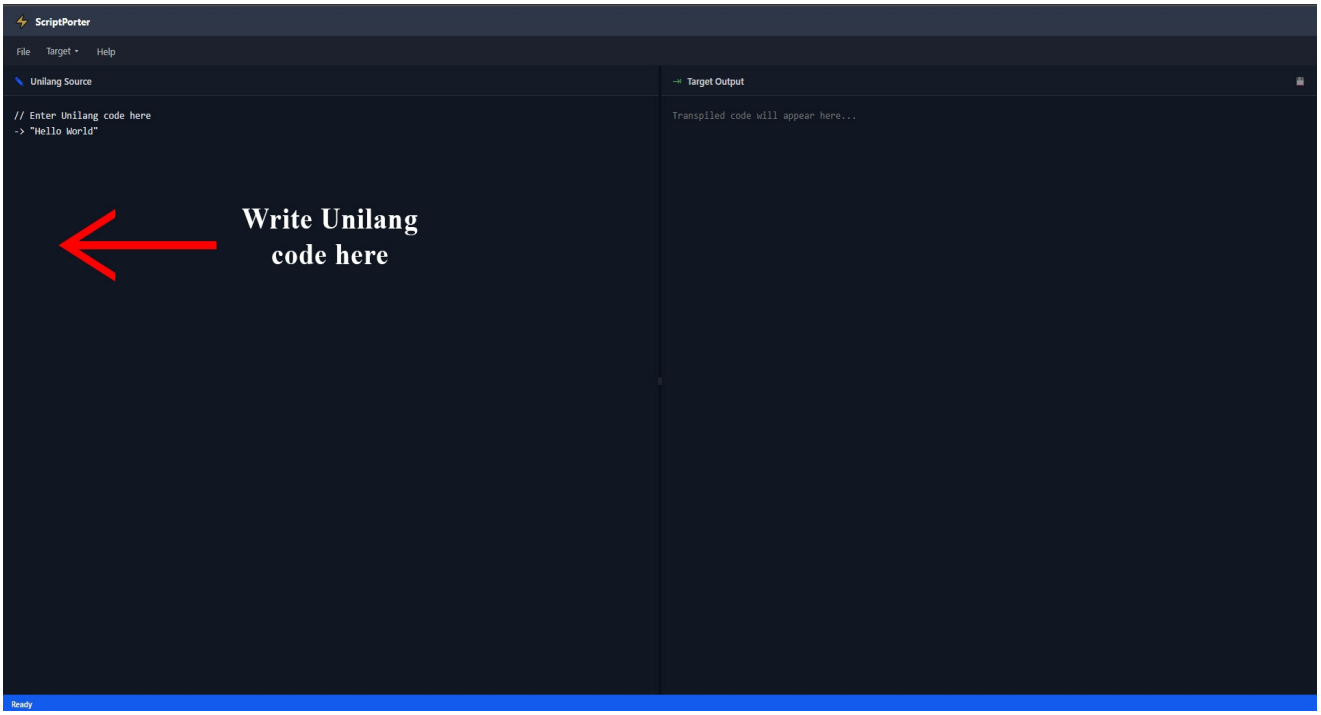


Help menu

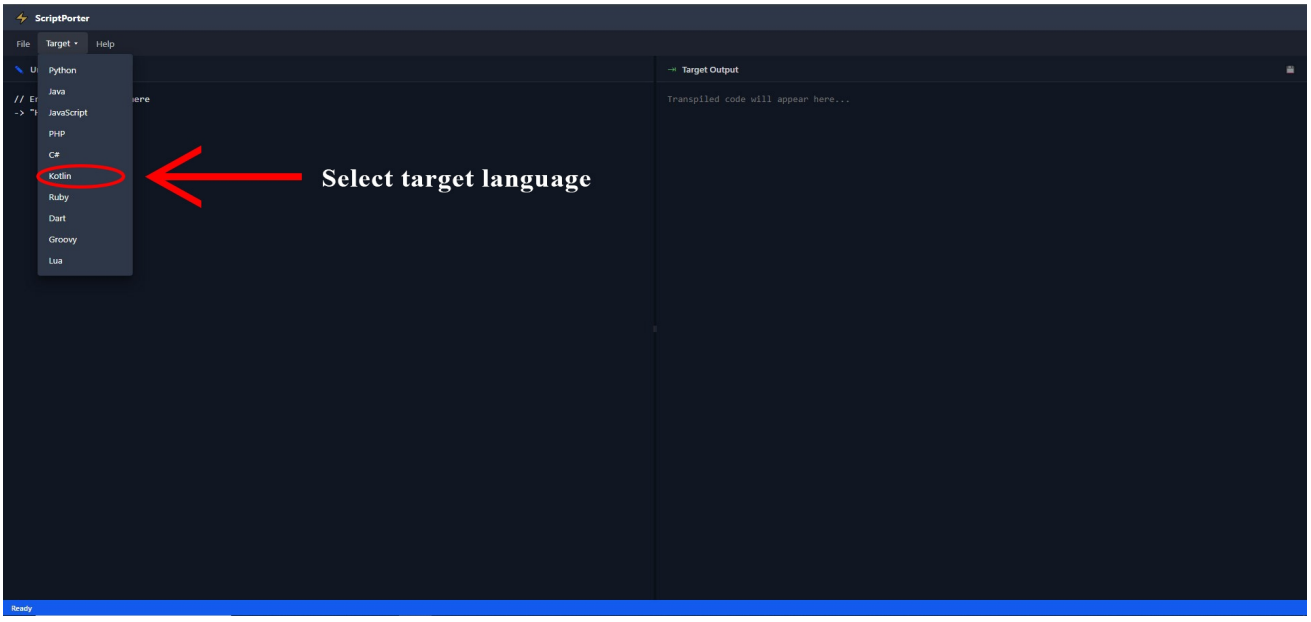


Working with ScriptPorter

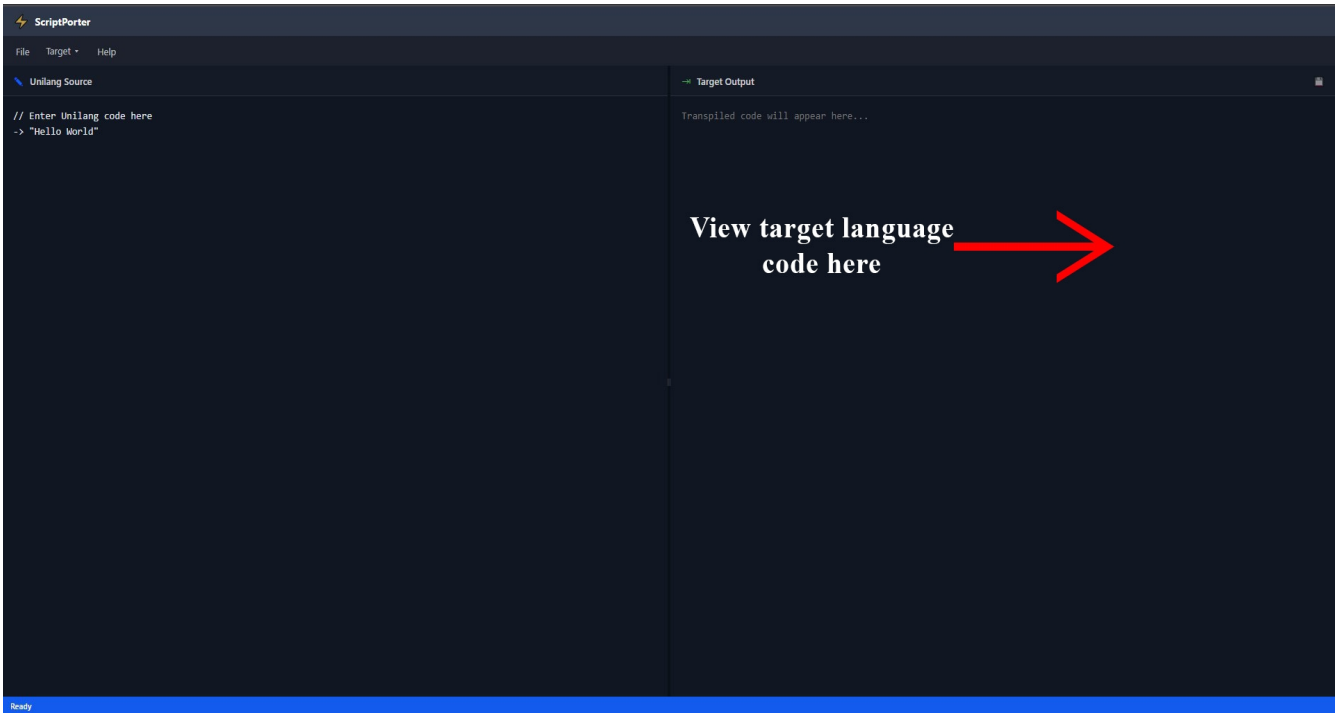
a) Write Unilang code



b) Select target language



c) View and save results in target language



UNILANG Documentation

ScriptPorter

UNILANG

INDEX

Introduction

Welcome to ScriptPorter Unilang— a custom-designed language created to simplify and streamline code translation between multiple general-purpose programming languages.

Modern software development often requires writing the same logic in multiple languages (e.g., Java, Python, JavaScript, C#, etc.) depending on the platform or target environment. This duplication introduces overhead, increases maintenance complexity, and leads to potential inconsistencies. ScriptPorter Unilang solves this by acting as a bridge — allowing developers to write logic once in a simple DSL and then translate it into the desired language.

Unilang's structure and syntax is very easy to learn even for a beginner. This documentation will guide you through the structure and syntax of Unilang. This will help you understand how to use and extend Unilang effectively.

***Notice:** The sample programs may contain comments (//), we recommend not using comments as they may affect the translation of some programs contain statements followed by comments. If you need to use comments place them in the next or previous line of a statement.

Keywords

Unilang uses few keywords compared to other langauges

out : Same as 'break'.

go : Same as 'continue'.

fix : Used to create static (read-only) variables.

ret : Same as 'return'.

all : Refers to all parameters in a function.

in : Same as 'in' in python

this : Same as 'this' keyword in C++ and java, 'self' in python

lock : Same as 'private'

flow : Same as 'protected'

unlock : Same as 'public'

Data Types & Variables

We can declare and initialize variables using '=' operator.

Here ScriptPorter Unilang automatically converts the variable to appropriate data type.

```
a=10            //integer
b="hello"       //string
```

Here data type is checked automatically. We can also specify the data type of a variable explicitly.

The various data types are:

(0)	:	Integer
(0.0)	:	Floating point value
(00)	:	Long
(0.00)	:	Double
(" ")	:	String
(n)	:	Number (Any numeric datatype)

```
e.g:  a=(0)          //integer
      b=(" ")        //string
      c=(0.0)        //Float
      d=(00)         //Long
```

Any statements within **" "** is considered as string.

We can create static variables using the keyword '**fix**' .

e.g: fix a=10 //declares 'a' as static variable

When using '**fix**' keyword the value of the variable can't be modified (constant).

It converts the variable to read-only variable.

Arrays and Lists

In Unilang, we can create arrays and list by placing elements within **[]**

e.g: arr1=[1,2,3]

Array (list) functions

(add): To insert elements to an array or list

e.g:

```
arr1=[]
(add:arr1,1)            //appends element 1 to array arr1
(add:arr1,2)
(add:arr1,2,3)          //inserts element 3 to index 2 of array arr1
```

(remove): To remove an element from an array or list

e.g:

```
arr1=[1,2,3,4,5]
(remove:arr1,3)             //removes element at index 3 (That is 4)
```

(replace): To replace an element with other

e.g:

```
arr1=[1,2,3,4,5]
(replace:arr1,2,7)      //replaces element at index 2 with 7
```

(len): To find size of an array or list

e.g:

arr1=[1,2,3,4,5]

->(len:arr1) //result is 5

(where): To find index (location) of an element in an array or list

e.g:

arr1=[1,2,3,4,5]

->(where:arr1,3) //the result is 2

Strings

A string is a sequence of characters. Anything contained in **“ “** is treated as string.

*****Notice:** Copy pasting of this sample codes are not recommended as they may cause format issues in the Unilang editor (like **” ”** and **""**).

e.g: str1=”hello”

String functions

(len): Returns size (length) of a string

e.g:

str1=”hello”

->(len:str1) //the result is 5

(up): Returns string in uppercase

e.g:

str1=”Hello”

->(up:str1) //HELLO

(low): Returns string in lowercase

e.g:

str1=”HELLO”

->(low:str1) //hello

I/O Operations

The '**->**' is the output operator

e.g: ->"Hello" //outputs 'Hello'

The '**<-**' is the input operator

e.g: a<- //reads input from user and assigns to variable 'a'

Sample Program 1

```
->"Enter a number:"
a<-                    //reads user input and assigns the value to 'a'
->"The number is", a
```

(You can use **a<-(0)** to read integer values and **a<-(0.0)** for float)

In this program the input from the user is printed. We can use '**,**' to concatenated output string.

Sample Program 2

```
a=10
->"The value is ", a
```

Operators

Arithmetic operators

+	Addition	Adds two values
-	Subtraction	Subtracts one value from other
*****	Multiplication	Multiplies two values
/	Division	Divides one value by other
%	Modulus	Returns remainder after division
****	Exponent	Returns power

e.g: a=10

b=2

->a+b //output '12' is printing to screen

->a-b //output is '8'

->a*b //output is '20'

->a/b //output is '5'

->a**b //output is '100'

->a%b //output is '0'

Relational operators

==	If the value of two operands are equal, then returns true
!=	If the value of two operands are not equal , then returns true

>	Returns true if the value of the left operand is greater than right operand
<	Returns true if the value of the left operand is less than right operand
>=	Returns true if the value of the left operand is greater than or equal to right operand
<=	Returns true if the value of the left operand is less than or equal to right operand

e.g: a=10

b=5

c=10

->a==b //output is 'false'

->a>b //output is 'true'

->a<b //output is 'false'

->a>=b //output is 'true'

->a<=b //output is 'false'

->a==c //output is 'true'

->a!=c //output is 'false'

Logical operators

&&	AND	exp1 && exp2	True only if both exp1 and

			exp2 are true; otherwise, false
||	OR	Exp1 || exp2	True if either exp1 or exp2 is true; false only if both are false
!	NOT	!exp1	False if exp1 is true; true if exp1 is false

Control Flow

Conditional statements

If

The '**?()**' can be used as 'if' statement in ScriptPorter Unilang.

Sytax:
```
?(condition)
{
    //statements
}
```

If the condition is true, then the statements contained in {} will be executed.

e.g:
```
value=5
?(value>0)
{
    ->"Positive number"
}
```

Here if the value is greater than 0 then prints the statement 'Positive number'. So the output is 'Positive number'.

If..else

The '**?()**' and '**$()**' is used as if and else statements.

```
Syntax:     ?(condition)                    //similar to 'if'
            {
                //true block statements
            }
            $()                             //similar to 'else'
            {
                //false block statements
            }
```

If..else if ..else

The '**??()**' can be used as the 'else if' statement.

```
Sytax:      ?(Condition 1)                  //if
            {
                //statements
            }                               //else if
            ??(Condition 2)
            {
                //statements
            }
            ??(Condition 3)                 //else if
            {
```

```
        //statements
    }
    $()                                    //else
    {
        //statements
    }
```

e.g:

```
a=5
?(a<0)
{
    ->"Negative number"
}
??(a==0)
{
    ->"Number is zero"
}
$()
{
    ->"Positive number"
}
```

Here the output is 'Positive number'

Switch statement

The '**…()**' can be used as 'switch' statement

Syntax:

```
…(test variable)
{
        Value1://statements
        Value2://statements
        ValueN://statements
}
```

e.g:

```
x=7
…(x)
{
        5: ->"Five"
                out
        6:->"Six"
                out
        7:"Seven"
                out
}
```

Here the value of 'x' is tested against a set of values. Here the output is 'Seven'. The keyword '**out**' is used to escape from the block, same as 'break' statement in other programming languages. We can use '**def**' to specify default case.

Loops

The '**()…**' can be used as both for and while loop. It can be used in the following ways:

1. (expression)…

```
{
    //statements
}
```

e.g:

```
x=1
(x<=10)…                    //same as 'while' loop
{
    ->x
    x++
}
```

Here the output is values 1 to 10

2. (start,end)…

```
{
        //statements
}

e.g:
        (val,11)…
        {
                ->val
                val++
        }
```

Start is 0 and the end value is excluded. Here the output is numbers from 0 to 10.

```
3.      (start, end, increment/decrement)…          //same as 'for' loop
        {
                //statements
        }

e.g:    (i=2,  11, 2)              //same as 'for' loop with increment 2
        {
                ->i
        }
```

Here the output is 2, 4, 6, 8, 10. The value of variable 'i' is initialized to 2. Then incremented by 2 upto value 11.

In the above mentioned loops, the body of the loop is only executed if the condition is true, otherwise the body will not be executed. If we need to execute the loop body at least once before checking the condition we can use '**…()…**' (In switch we use **…()** but here **…()…**). It is similar to 'do' statement in other programming languages.

Syntax: **…(expression)…**

{

//statements

}

e.g: i=11

…(i<10)… // 'while(i<10)'

{

->i

}

Here the output is 11.

If we are not using '**…()…**' statement, the loop body will not be executed here because the value of 'i' is 11. But when using '**…()…**' the body is executed for the first time without checking the condition. Then checks the condition. If it is true, the statements are executed again.

The keyword '**out**' is similar to 'break' in other programming languages and '**go**' is used as 'continue'.

Functions

A function is a block of code to perform a specific task. The function body is contained in **{}.** Functions are called by **(function_name)**.

Without arguments

Syntax : function_name<-{

//statements

}

e.g: greet<-{

->"Hello"

}

The function 'greet' is called from main function by:

(greet)

So the output is 'Hello'

With arguments

Syntax: function_name<-{

//function body

}<-arg1,arg2

Here 'arg1' and 'arg2' are arguments passed to the function from main function.

```
e.g:   sum<-{
                    ->x+y
            }<-x,y

            (sum:5,10)
```

Here the function 'sum' is called and passing the parameters 5 and 10. The function 'sum' takes two arguments 'x' and 'y', it represents the values passed. Here the output is '15'.

With return value

We can use the '**ret'** keyword to return value from a function, it is same as 'return' keyword in other languages. When we use functions with '**ret**' we need a variable in the main function to store the value returned.

```
e.g:   sum<-{
                    ret a+b
            }<-a,b
```

```
res=(sum:2,8)
->res
```

Here the output is 10. The value returned from the function is assigned to variable 'res'.

Sample Programs

Here is a program that performs various arithmetic operations using functions (function with arguments, without arguments, with return, etc)

```
Addition<-{                          //Function without arguments
        X=5
        Y=10
        ->"The sum =", X+Y
}

Subtraction<-{                       //Function with arguments
            ->"Result=", a-b
        }<-a,b
```

```
Division<-{                              //Function with return
            ret x/y
        }<-x,y

Multiplication<-{                        //Function with return
                ret a*b
            }<-a,b

(Addition)                  //calls function 'Addition'
(Subtraction:10, 2)         //calls 'Subtraction'
div=(Division:8, 2)         //calls 'Division' function
mul=(Multiplication:5,2)    //Multiplication
->"The quotient is ",div
->"The product is ",mul
```

```
The output is:  'The sum=15
                Result=8
                The quotient is 4
                The product is 10'
```

Default arguments

We can assign default values to function arguments. If no value is passed as argument, the default value is used.

```
e.g:   sum<-{
                    ->x+y
            }<-x,y=10                    //the default value of 'y' is 10

       (sum:10)          //passing only one parameter
```

Here no value is passed for argument 'y' from the main function. So the value is defaulted to 10 (the default value of argument 'y'). So the result is 20.

Classes

The class body is contained in **[ ]**

Syntax: classname<-[//variables & functions]

e.g : person<-[

```
name="Noname"
age=50

print<-{
        ->"Name=",this.name,
          "Age=",this.age
    }
]
```

The class contains variables and functions. Class objects are used to access class functions and variables. The '**.**' (dot) operator is used to access class variables and methods.

We use '**this**' keyword to refer to class variables within class functions.

In the above example, a class 'person' is created. We can create object of that class by:

P1=person()

Here the we declare 'P1' as an object of class 'person' , so P1 can access variables (name, age) and functions (print) of the class 'person'

```
->P1.name                    //output is 'Noname'
->P1.age                     //output is '50'
->P1.(print)                 //output is 'Name=Noname Age=50'
```

Class with default constructor

We can initialize class variables without assigning values using objects.

```
Syntax:    classname<-[
                    Variable1=value1      //default constructor
                    Variable2=value2
                    ]
```

```
e.g:  person<-[
                  name="Noname"
                  age=100
                  place="noplace"
```

```
                ]
p=person()
->p.name
->p.age
->p.place
```

Here we just access the values of the class variables using object of that class. We are not assigning values but just using the values.

Inheritance

We use '**@**' to derive a class from another class.

Single inheritance

```
Syntax:     Class1<-[                       //The base class
                ]
            Class2@Class1<-[                //The derived class
                            ]
```

Here **Class2** is derived from **Class1** and has access to the members of class **Class1. Class2** gets properties of **Class1.**

Multiple inheritance

A class derived from more than one class.

Syntax: class3@class1,class2<-[//variables & functions

]

Here **class3** gets properties of **class2** and **class1.** Multiple classes can be separated by commas(,).

Multi-level inheritance

One class is derived from a class that is derived from another class.

Syntax: Class1<-[ ]

Class2@Class1<-[ ]

Class3@Class2<-[ ]

Here **Class1** is the base class. **Class2** is derived from **Class1** and **Class3** is derived from **Class2**. So **Class2** gets properties of **Class1**. **Class3** gets properties of **Class2** and **Class1**.

Hierarchichal inheritance

A class is used as base class for many derived classes.

Syntax: class1<-[ ]

class2@class1<-[ ]

class3@class1<-[ ]

Here **class1** serves as the base class for **class2** & **class3**. Both **class2** and **class3** are derived from same base class **class1**.

Access Specifiers

The access specifiers defines scope and accessibility of class variables and functions. By default it is '**unlock**'.

lock : Accessible only within the class. Can't be accessed from outside the class. Same as 'private'.

flow : Accessible within the class and derived class. Same as 'protected'.

unlock: Accessible anywhere in the program. Same as 'public'.

Exception Handling

We can place the statements that may generate a runtime error in the **?{ }** block and the statements to manage that error in the **${ }** block. It may look similar to **?()** (if) and **$()** (else) , except **()**.

Syntax:
```
?{
        //statements that may generate an error
}
${
        //statements to manage the error
}
```

e.g:
```
?{
        ->"Enter a number:"
        a<-(0)                  //Expects integer value
}
${
        ->"Enter integers only"
}
```

Here we expect an integer value for variable ‘a’. If the user enters a non-integer value, the statements in the **${}** block is executed. For example: If the user enters the value 5, no error occurs. If the user enters the value 5.1 then a type mismatch error occurs. So the statements in the **${}** block is executed, that is 'Enter integers only'.

We don't need to specify the type of error, the exception type is automatically checked. But we need to place enough **${}** to handle each exception.

Built-in Functions

(get) : Used to import packages (works only with python).

Syntax: (get:package_name)

e.g: pack1=(get:math) //imports 'math' package

(len) : Returns size of array, list, etc

Syntax: (len:array)

e.g: size=(len:array1) //size of array1 is assigned to ‘size’

(add) : Inserts an element to an array

Syntax: (add:array,index,element) //insert element at the specified position

(add:array,element) //appends element to the array

e.g: add(array1,2,5) //insert element 5 at index 2 of array1

(remove) : Removes an element from an array or list

Syntax: (remove:array,index)

e.g: (remove:array1,2) //removes item with index 2 from array1

(replace) : Replaces one element with other

Syntax: (replace:array,index_of_element_to_replace,new_element)

e.g: (replace:array1,1,5) //replaces element at index 1 with 5

(where) : Returns index of an element

Syntax: (where:array,element_to_find)

e.g:(where:array1,3) //returns index of element 3 in array1

(up) : Returns a string in uppercase

Syntax: (up:string)

e.g: str1=”Testing”

->(up:str1) //returns str1 in uppercase

(low) : Returns a string in lowercase

Syntax: (low:string)

e.g: str1=”Testing”

->(low:str1) //returns str1 in lowercase

Appendices

sample programs

***Notice**: We recommend not using comments, as they may affect the translation of some programs. If you want to use comments, place them in the next or previous line of a statement.

1. Program to perform basic input output operation

```
->"Enter your name:"
name<-
->"Welcome",name
```

2. Program to read two numbers and print sum

```
->"Enter two numbers:"
a<-(0)
b<-(0)
->"Sum=",a+b
```

3. Program to read a number and find whether it is a positive or negative number

```
->"Enter a number:"
```

```
a<-(0)

?(a<0)
{
   ->"Negative number"
}

??(a==0)
{
   ->"Zero"
}

$()
{
   ->"Positive number"
}
```

4. Program to print numbers from 1 to 10 using while loop, for loop and do..while loop

```
i=1
(i<=10)...
{
   ->i
   i=i+1
}

(j=1,11,1)...
{
   ->j
}

k=1
...(k<=10)...
{
   ->k
   k=k+1
}
```

5. Program to read and print an array

```
arr1=[]
->"Enter no.of elements:"
no<-(0)

(i=0,no,1)...
{
   ->"Enter element to insert:"
   x<-(0)
   (add:arr1,x)
}

->"The array is: ",arr1
```

6. Program to print odd and even numbers within given limits

```
->"Enter lower limit:"
start<-(0)
->"Enter upper limit:"
end<-(0)

odd=[]
```

```
even=[]

(start<=end)...
{
   ?(start%2==0)
   {
      (add:even,start)
   }
   $()
   {
      (add:odd,start)
   }

   start=start+1
}

->"Odd numbers: ",odd
->"Even numbers: ",even
```

7. Program using function

```
greet<-{
   ->"Hello"
}

sum<-{
   ->"The sum=",x+y
}<-x,y

(greet)
(sum:5,10)
```

8. Program to use string handling functions

```
str1="Testing"
->(up:str1)
->(low:str1)
```

9. Program using switch statement

```
-> "Enter a number (1, 2, or 3):"
temp <-(0)
...(temp)
```

```
{
  1:
    -> "You entered One"
  2:
    -> "You entered Two"

  3:
    -> "You entered Three"
  def:
    ->"Invalid option!"
}
```

10. Program to perform various mathematical operations using functions (with and without return)

```
addition<-{
        ->"Sum=",x+y
}<-x,y

subtraction<-{
        ->"Difference=",x-y
```

```
}<-x,y

multiplication<-{
   ret x*y
}<-x,y

division<-{
   ret x/y
}<-x,y



->"Enter two numbers:"
a<-(0)
b<-(0)

(addition:a,b)
(subtraction:a,b)
->"Product=",(multiplication:a,b)
->"Quotient=",(division:a,b)
```

11. Program to demonstrate exception handling

```
?{
   ->"Enter a number:"
   a<-(0)
   ->"The number is ",a
}

${
   ->"Integers only!"
}
```

12. Program using class

```
person<-[
   Name="John"
   Age=50
]

p1=person()
->p1.Name
->p1.Age
```

13. Program using inheritance

```
vehicle<-[
   Wheels=0
   Speed=0

   display<-{
      ->"Wheels=",this.Wheels,"Speed=",this.Speed
   }
]

car@vehicle<-[
   Wheels=4
   Speed=200
]
c1=car()
c1.(display)
```

14. Program for using various list (array) handling functions

```
arr1=[]
```

```
->"Enter no.of elements:"
n<-(0)
->"Enter elements:"

(i=0,n,1)...
{
   x<-(0)
   (add:arr1,i,x)  //Adding elements
}

->"The array is:",arr1

s=(len:arr1)       //size of the array
->"Length of the array=",s

->"Enter index of element to be removed:"
rm<-(0)
(remove:arr1,rm)   //remove element
->"Array after removing elements:",arr1
```

```
(replace:arr1,1,5)  //replacing element
->"Array after replaceing second element with 5:",arr1


->"Enter element to find:"
el<-(0)


loc=(where:arr1,el) //finding index of element
->"The element ",el," is at ",loc


->"Enter element to add:"
num<-(0)
(add:arr1,num)
->"Array after adding element:",arr1
```

15. Program for Fizz Buzz game

```
(i=1,31,1)...
{
   ?(i%3==0 && i%5==0)
   {
      ->"fizz buzz"
```

```
    }
    ??(i%3==0)
    {
        ->"fizz"
    }
    ??(i%5==0)
    {
        ->"buzz"
    }
    $()
    {
        ->i
    }
}
```