v0.0.1 HCPL Code Structure

Variables

It's used to store different values such as numbers, strings, or other data types.

Syntax1:

```
let VARIABLE_NAME be VALUE
```

Example1:

```
let x be 10
display x
```

output:

```
10
```

Syntax2:

```
let VARIABLE_NAME1, VARIABLE_NAME2, ... VARIABLE_NAMEn, be VALUE1, VALUE2, ... VALUEn
```

Example2:

```
let a, b, c be 10, 20, 30
display a
display b
display c
```

output:

```
10
20
30
```

Variable Concatenation

Combine two or more variables into a single output. This works for both numeric and string variables.

Syntax:

```
display VARIABLE_NAME1 + VARIABLE_NAME2
```

Example1:

```
let x be 10
let y be 5
display x + y
```

output:

```
15
```

Example2:

```
let x be 'Hello '
let y be 'World'
display x + y
```

output:

```
Hello World
```

v0.0.1 HCPL Code Structure

Variable Types

Used to check the data type of a variable.

Syntax:

```
what is type of VARIABLE_NAME
```

Example1:

```
let x be 10
what is type of x
```

output:

```
i
```

Example2:

```
let y be 'swagat'
what is type of y
```

output:

```
str
```

Example3:

```
let a be true
what is type of a
```

output:

```
b
```

Example4:

```
let b be 5.5
what is type of b
```

output:

```
f
```

Example5:

```
let c be 10.00
what is type of c
```

output:

```
d
```

Example6:

```
let d be 'c'
what is type of d
```

output:

```
C
```

v0.0.1 HCPL Code Structure

Variable Rules

The name of a variable must start with a letter and not a number.

Syntax:

```
let VARIABLE_NAME be VALUE
```

Example1:

```
let 1a be 10
#This is invalid variable.
```

output:

```
Error
```

Example2:

```
let a1 be 10
#This is valid variable
```

output:

```
10
```

Comments

Used to explain code or prevent execution. Single-line comments are used for short explanations, while multi-line comments can span several lines.

Syntax1:

```
#This is a single line comment
```

Syntax2:

```
'''
This is Multi line
comment
'''
```

Data Types

Categorizes the kind of data a variable can hold.

1. Text Type: str
2. Numeric Type: int, float
3. Sequence Type: list, tuple, range
4. Boolean Type: bool
5. Mapping Type: dict

v0.0.1 HCPL Code Structure

Variable Casting

Converts a value from one data type to another.

Syntax:

```
let VARIABLE_NAME be (TYPE: VALUE)
```

Example1:

```
let x be (float: 1)
display x
```

output:

```
1.0
```

Example2:

```
let y be (str: 69)
display y
```

output:

```
'69'
```

String

It contains a sequence of characters enclosed in single, double, or triple quotes.

Syntax:

```
let VARIABLE_NAME be '<STRING_STATEMENT>'
```

Example1:

```
let x be 'This is a string of words that will be stored in variable x.'
display x
```

output:

```
This is a string of words that will be stored in variable x.
```

Example2:

```
let x be “““Rohan is sick everytime so, he have to do meditation, and exercise daily.”””
display x
```

output:

```
Rohan is sick everytime so, he have to do meditation, and exercise daily.
```

Slicing String

Extracts a specific part of a string using a range of indexes.

Syntax:

```
VARIABLE_NAME
start:end
```

Example:

```
let x be 'Hello World'
display x[6:9]
display x[:5]
display x[2:]
display x[−4:−2]
```

output:

```
Wor
Hello
llo World
Orl
```

String Modification

Transforms a string into a new format.

Syntax:

```
display VARIABLE_NAME as METHOD_NAME
```

Example:

```
let x be 'Manish Mhatre'
display x as uppercase
display x as lowercase
display x as strip
display x as split using ','
```

output:

```
MANISH MHATRE
manish mhatre
ManishMhatre
′Manish′,′Mhatre′
```

v0.0.1 HCPL Code Structure

Formatted String

Allows you to insert variables directly into a string.

Syntax:

```
let formatted string VARIABLE_NAME be 'String with {VARIABLE_NAME}'
```

Example:

```
let age be 21
let formatted string x be 'My name is Manish, I am {age} years old.'
display x
```

output:

```
My name is Manish, I am 21 years old.
```

Escape Character

Used to insert special characters that cannot be typed directly.

Syntax:

\n for new line, \t for tab

Example1:

```
let x be """Rohan is sick everytime \n so, he have to do meditation, and exercise daily."""
display x
```

output:

```
Rohan is sick everytime
so, he have to do meditation, and exercise daily.
```

Example2:

```
let x be 'H \t e \t l \t l \t o'
display x
```

output:

```
H	e	l	l	o
```

v0.0.1 HCPL Code Structure

String Methods

Built-in functions for manipulating strings.

- **Capitalize:**
  Converts the first character to uppercase.
  Syntax:
  display VARIABLE_NAME as capitalize
  Example:
  let x be 'hello world'
  display x as capitalize
  output:
  HELLO WORLD

- **Casefold:**
  Converts string to lowercase.
  Syntax:
  display VARIABLE_NAME as casefold
  Example:
  let x be 'Hello World'
  display x as casefold
  output:
  hello world

- **Center:**
  Pads the string with spaces to a certain width.
  Syntax:
  display VARIABLE_NAME as center
  Example:
  let x be 'hello'
  display x as center
  output:
  " hello "
- **Count:**
  Returns the number of times a specified value occurs.
  Syntax:
  display VARIABLE_NAME as count using 'STRING_OR_CHARACTER'
  Example:
  let x be 'apple banana apple'
  display x as count using 'apple'
  output:
  2

- **Expandtab:**
  Replaces tab characters with spaces.
  Syntax:
  ```
  display VARIABLE_NAME as expandtab in NUMBER
  ```
  Example:
  ```
  let x be 'H\te\tl\tl\to'
  display x as expandtab in 2
  ```
  output:
  ```
  H          e          l          l          o
  ```

- **Index Find:**
  Searches for a specified value and returns its position.
  Syntax:
  ```
  display VARIABLE_NAME as index using 'STRING_OR_CHARACTER'
  ```
  Example:
  ```
  let x be 'Hello world'
  display x as index using 'world'
  ```
  output:
  ```
  6
  ```

- **Join:**
  Combines all elements of an iterable into a string.
  Syntax:
  ```
  let VARIABLE_NAME1 be 'SEPARATOR' as join using VARIABLE_NAME2
  ```
  Example:
  ```
  let x be ('Swagat', 'Manish', 'Sarvesh')
  let y be '#' as join using x
  display y
  ```
  output:
  ```
  Swagat#Manish#Sarvesh
  ```

v0.0.1 HCPL Code Structure

Boolean Values

Represents one of two values: **true** or **false**.

Syntax:

```
bool as VALUE
```

Example1 - Only true output:

```
bool as 'abc'
bool as 123
```

Example2 - Only false output:

```
bool as none
bool as 0
bool as ' '
bool as
bool as {}
```

Operators

Symbols that perform operations on values and variables.

- **Assignment**

  Assignment operators are used to assign values to variables. Assigns a value from the right-hand operand to the left-hand operand.

  Syntax 1:

  ```
  let VARIABLE_NAME be VALUE
  ```

  Syntax 2:

  ```
  VARIABLE_NAME = VALUE
  ```

  Example 1:

  ```
  let x be 10
  ```

  Example 2:

  ```
  x = 10
  ```

- **Logical**

  Logical operators are used to determine the logic between variables or values.

➢ **and**

Definition: Returns true if both statements are true.

Syntax:

```
CONDITION1 and CONDITION2
```

Example:

if this x > 5 and x < 10 happen then…

➢ **or**

Definition: Returns true if one of the statements is true.

Syntax:
CONDITION1 or CONDITION2
Example:
if this x == 5 or y == 5 happen then...

➢ **not**

Definition: Reverses the result, returns false if the result is true.

Syntax:
not CONDITION
Example:
if this not x > 5 happen then...

- **Arithmetic**

Arithmetic operators are used to perform mathematical operations.

➢ **+ (Addition)**

Definition: Adds two values together.

Syntax 1:
add VARIABLE_NAME to VALUE
Example 1:
add x to 2
Syntax 2:
VARIABLE_NAME = VARIABLE_NAME + VALUE
Example 2:
x = x + 2

➢ - (Subtraction)

Definition: Subtracts one value from another.

Syntax 1:
subtract VARIABLE_NAME to VALUE
Example 1:
subtract x to 2
Syntax 2:
VARIABLE_NAME = VARIABLE_NAME - VALUE
Example 2:
x = x - 2

➢ * (Multiplication)

Definition: Multiplies two values.

Syntax 1:
multiply VARIABLE_NAME to VALUE
Example 1:
multiply x to 2
Syntax 2:
VARIABLE_NAME = VARIABLE_NAME * VALUE
Example 2:
x = x * 2

➢ / (Division)

Definition: Divides one value by another.

Syntax 1:
divide VARIABLE_NAME to VALUE
Example 1:
divide x to 2
Syntax 2:
VARIABLE_NAME = VARIABLE_NAME / VALUE
Example 2:
x = x / 2

v0.0.1 HCPL Code Structure

➢ % (Modulo)

Definition: Returns the remainder of a division.

Syntax 1:
mod VARIABLE_NAME to VALUE
Example 1:
mod x to 2
Syntax 2:
VARIABLE_NAME = VARIABLE_NAME % VALUE
Example 2:
x = x % 2

- **Comparison**

Comparison operators are used to compare two values.

➢ == (Equals)

Definition: Returns true if the values are equal.
Syntax 1:
VARIABLE_NAME equals to VALUE
Example 1:
if this x equals to 2 happen then...
Syntax 2:
VARIABLE_NAME == VALUE
Example 2:
if x == 2 then...

➢ != (Not Equals)

Definition: Returns true if the values are not equal.
Syntax 1:
VARIABLE_NAME not equals to VALUE
Example 1:
if this x not equals to 2 happen then...
Syntax 2:
VARIABLE_NAME != VALUE
Example 2:
if x != 2 then...

v0.0.1 HCPL Code Structure

➢ > (Greater Than)

Definition: Returns true if the first value is greater than the second.

Syntax 1:

VARIABLE_NAME more than to VALUE

Example 1:

if this x more than to 2 happen then...

Syntax 2:

VARIABLE_NAME > VALUE

Example 2:

if x > 2 then...

➢ < (Less Than)

Definition: Returns true if the first value is less than the second.

Syntax 1:

VARIABLE_NAME less than to VALUE

Example 1:

if this x less than to 2 happen then...

Syntax 2:

VARIABLE_NAME < VALUE

Example 2:

if x < 2 then...

➢ >= (Greater Than or Equal to)

Definition: Returns true if the first value is greater than or equal to the second.

Syntax 1:

VARIABLE_NAME more than or equal to VALUE

Example 1:

if this x more than or equal to 2 happen then...

Syntax 2:

VARIABLE_NAME >= VALUE

Example 2:

if x >= 2 then...

➢ **<= (Less Than or Equal to)**

Definition: Returns true if the first value is less than or equal to the second.

Syntax 1:

VARIABLE_NAME less than or equal to VALUE

Example 1:

if this x less than or equal to 2 happen then...

Syntax 2:

VARIABLE_NAME <= VALUE

Example 2:

if x <= 2 then...

- **Identity**

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location.

➢ **is**

Definition: Returns true if both variables are the same object.

Syntax:

VARIABLE_NAME1 is VARIABLE_NAME2

Example:

```
let x be 10
let y be 10
if x is y then display 'x and y are the same object.'
```

➢ **is not**

Definition: Returns true if both variables are not the same object.

Syntax:

VARIABLE_NAME1 is not VARIABLE_NAME2

Example:

```
let list1 be 'apple', 'banana'
let list2 be 'apple', 'banana'
if list1 is not list2 then
display 'list1 and list2 are not the same object.'
```

v0.0.1 HCPL Code Structure

Input Function

Gets user input from the console.

Syntax:

```
input VARIABLE_NAME with prompt 'String'
```

Example:

```
input x with prompt 'Enter your name: '
```

output:

```
Enter your name: (user input here)
```

Array (List)

An ordered and changeable collection of items.

Syntax:

```
let VARIABLE_NAME be
′VALUE1′,′VALUE2′,...,′VALUEn′
```

Example:

```
let x be
′value1′,′value2′,′value3′
display x
```

output:

```
′value1′,′value2′,′value3′
```

Array Methods

Built-in functions for manipulating arrays.

- **append:**

  Adds an element to the end of the array.

  Syntax:

  ```
  VARIABLE_NAME as append('VALUE')
  ```

  Example:

  ```
  let x be′value1′,′value2′
  display x as append('value3')
  ```

  output:

  ```
  ′value1′,′value2′,′value3′
  ```

- **clear:**
  Removes all elements from the array.
  Syntax:
  VARIABLE_NAME as clear
  Example:
  let x be′value1′,′value2′
  display x as clear
  output:

- **copy:**
  Returns a copy of the array.
  Syntax:
  let VARIABLE_NAME2 be VARIABLE_NAME1 as copy
  Example:
  let x be′value1′,′value2′
  let y be x as copy
  display y
  output:
  ′value1′,′value2′

- **countoflist:**
  Returns the number of elements with the specified value.
  Syntax:
  display VARIABLE_NAME as countoflist('VALUE')
  Example:
  let x be′value1′,′value2′,′value1′
  display x as countoflist('value1')
  output:
  2

- **extends:**
  Adds the elements of an iterable to the end of the current array.
  Syntax:
  VARIABLE_NAME1 as extends(VARIABLE_NAME2)
  Example:
  let x be′value1′,′value2′
  let y be′value3′,′value4′
  display x as extends y
  output:
  ′value1′,′value2′,′value3′,′value4′

v0.0.1 HCPL Code Structure

- **index:**
  Returns the index of the first element with the specified value.
  Syntax:
  display VARIABLE_NAME as index('VALUE')
  Example:
  let x be′value1′,′value2′,′value3′
  display x as index('value2')
  output:
  1

- **insert:**
  Adds an element at the specified position.
  Syntax:
  VARIABLE_NAME as insert(POSITION, 'VALUE')
  Example:
  let x be′value1′,′value3′
  display x as insert(1, 'value2')
  output:
  ′value1′,′value2′,′value3′

- **pop:**
  Removes the element at the specified position.
  Syntax:
  display VARIABLE_NAME as pop(POSITION) or display VARIABLE_NAME as pop
  Example1:
  let x be′value1′,′value2′,′value3′
  display x as pop(1)
  output:
  'value2'
  Example2:
  let y be′value1′,′value2′,′value3′
  display y as pop
  output:
  'Value3'

- **remove:**
  Removes the first item with the specified value.
  Syntax:
  VARIABLE_NAME as remove('VALUE')
  Example:
  let x be′value1′,′value2′,′value1′
  display x as remove('value1')
  output:
  ′value2′,′value1′

- **reverse:**
  Reverses the order of the array.
  Syntax:
  VARIABLE_NAME as reverse
  Example:
  let x be′value1′,′value2′,′value3′
  display x as reverse
  output:
  ′value3′,′value2′,′value1′

- **sort:**
  Sorts the array.
  Syntax:
  VARIABLE_NAME as sort
  Example:
  let x be′value3′,′value1′,′value2′
  display x as sort
  output:
  ′value1′,′value2′,′value3′

v0.0.1 HCPL Code Structure

Set

An unordered, unchangeable, and unindexed collection. It does not allow duplicate values.

Syntax:

```
let VARIABLE_NAME be {'VALUE1', 'VALUE2', ..., 'VALUEn'}
```

Example:

```
let x be {true, false, 1, 2}
let y be {true, false, 0, 2}
display x
display y
```

output:

```
{true, false, 1, 2}
{true, false, 0, 2}
```

Set Constructor

Used to create a new set.

Syntax:

```
let VARIABLE_NAME be set((VALUE1, VALUE2, VALUE3))
```

Example:

```
let x be set (('value1', 'value2', 'value3'))
display x
```

output:

```
{'value1', 'value2', 'value3'}
```

Set Methods

Built-in functions for manipulating sets.

- **add:**

  Adds an element to the set.

  Syntax:

  ```
  VARIABLE_NAME as add('VALUE')
  ```

  Example:

  ```
  let x be {'value1', 'value2'}
  display x as add('value3')
  ```

  output:

  ```
  {'value1', 'value2', 'value3'}
  ```

- **copy:**
  Returns a copy of the set.
  Syntax:
  ```
  let VARIABLE_NAME2 be VARIABLE_NAME1 as copy
  ```
  Example:
  ```
  let x be {'value1', 'value2'}
  let y be x as copy
  display y
  ```
  output:
  ```
  {'value1', 'value2'}
  ```

- **clear:**
  Removes all elements from the set.
  Syntax:
  ```
  VARIABLE_NAME as clear
  ```
  Example:
  ```
  let x be {'value1', 'value2'}
  display x as clear
  ```
  output:
  ```
  {}
  ```

- **discard:**
  Removes the specified item.
  Syntax:
  ```
  VARIABLE_NAME as discard('VALUE')
  ```
  Example:
  ```
  let x be {'value1', 'value2'}
  display x as discard('value2')
  ```
  output:
  ```
  {'value1'}
  ```

- **intersect:**
  Returns a new set with common items.
  Syntax:
  ```
  display VARIABLE_NAME1 as intersect(VARIABLE_NAME2)
  ```
  Example:
  ```
  let x be {'value1', 'value2', 'value3'}
  let y be {'value3', 'value4', 'value5'}
  display x as intersect(y)
  ```
  output:
  ```
  {'value3'}
  ```

- **pop:**
  Removes a random element from the set.
  Syntax:
  ```
  display VARIABLE_NAME as pop
  ```
  Example:
  ```
  let x be {'value1', 'value2', 'value3'}
  display x as pop
  ```
  output:
  ```
  'value1' # (Output may vary as sets are unordered)
  ```

- **remove:**
  Removes the specified element.
  Syntax:
  ```
  VARIABLE_NAME as remove('VALUE')
  ```
  Example:
  ```
  let x be {'value1', 'value2'}
  display x as remove('value2')
  ```
  output:
  ```
  {'value1'}
  ```

- **union:**
  Returns a new set with all items from both sets.
  Syntax:
  ```
  display VARIABLE_NAME1 as union(VARIABLE_NAME2)
  ```
  Example:
  ```
  let x be {'value1', 'value2'}
  let y be {'value3', 'value4'}
  display x as union(y)
  ```
  output:
  ```
  {'value1', 'value2', 'value3', 'value4'}
  ```

- **update:**
  Adds the items from another set to the current set.
  Syntax:
  VARIABLE_NAME1 as update(VARIABLE_NAME2)
  Example:
  let x be {'value1': 'value1'}
  let y be {'value2': 'value2'}
  x as update(y)
  display x
  output:
  {'value1': 'value1', 'value2': 'value2'}

Tuple

An ordered and unchangeable collection of items.

Syntax:
let VARIABLE_NAME be ('VALUE1', 'VALUE2', ..., 'VALUEn')

Example1:
let x be ('value1', 'value2', 'value3', 'value4', 'value5', 'value6')
display x[2:5]
output:
('value3', 'value4', 'value5')

Example2:
let y list x
let y
1
be 'Value8'
let x be tuple y
display x
output:
('value1', 'Value8', 'value3', 'value4', 'value5', 'value6')

Tuple Constructor

Used to create a new tuple.

Syntax:

```
let VARIABLE_NAME be tuple((VALUE1, VALUE2, VALUE3))
```

Example:

```
let x be tuple (('value1', 'value2', 'value3'))
display x
```

output:

```
('value1', 'value2', 'value3')
```

Tuple Methods

Built-in functions for manipulating tuples.

- **countoftuple:**
  Returns the number of times a value appears.
  Syntax:
  ```
  display VARIABLE_NAME as countoftuple('VALUE')
  ```
  Example:
  ```
  let x be ('apple', 'banana', 'apple')
  display x as countoftuple('apple')
  ```
  output:
  ```
  2
  ```

- **index:**
  Searches for a value and returns its position.
  Syntax:
  ```
  display VARIABLE_NAME as index('VALUE')
  ```
  Example:
  ```
  let x be ('apple', 'banana', 'cherry')
  display x as index('banana')
  ```
  output:
  ```
  1
  ```

Dictionary

An unordered, changeable, and indexed collection of key-value pairs.

Syntax:

```
let VARIABLE_NAME be {'KEY1': 'VALUE1', 'KEY2': 'VALUE2'}
```

Example:

```
let x be {'key1': 'value1', 'key2': 53, 'key2': 2020}
display x['key2']
```

output:

```
2020
```

Dictionary Methods

Built-in functions for manipulating dictionaries.

- **clear:**

  Removes all elements from the dictionary.

  Syntax:

  ```
  VARIABLE_NAME as clear
  ```

  Example:

  ```
  let x be {'key1': 'value1', 'key2': 'value2'}
  display x as clear
  ```

  output:

  ```
  {}
  ```

- **copy:**

  Returns a copy of the dictionary.

  Syntax:

  ```
  let VARIABLE_NAME2 be VARIABLE_NAME1 as copy
  ```

  Example:

  ```
  let x be {'key1': 'value1', 'key2': 'value2'}
  let y be x as copy
  display y
  ```

  output:

  ```
  {'key1': 'value1', 'key2': 'value2'}
  ```

- **fromkeys:**
  Returns a dictionary with the specified keys and value.
  Syntax:
  ```
  let DICTIONARY_NAME be dict as fromkeys(LIST_OF_KEYS, DEFAULT_VALUE)
  ```
  Example:
  ```
  let x be ('key1', 'key2', 'key3')
  let y be dict as fromkeys(x, 0)
  display y
  ```
  output:
  ```
  {'key1': 0, 'key2': 0, 'key3': 0}
  ```

- **get:**
  Returns the value for a specified key.
  Syntax:
  ```
  display VARIABLE_NAME as get('KEY')
  ```
  Example:
  ```
  let x be {'key1': 'value1', 'key2': 'value2'}
  display x as get('key2')
  ```
  output:
  ```
  'value2'
  ```

- **items:**
  Returns a list of key-value pairs as tuples.
  Syntax:
  ```
  display VARIABLE_NAME as items
  ```
  Example:
  ```
  let x be {'key1': 'value1', 'key2': 'value2'}
  display x as items
  ```
  output:
  ```
  (′key1′,′value1′),(′key2′,′value2′)
  ```

- **key:**
  Returns a list of the dictionary's keys.
  Syntax:
  ```
  display VARIABLE_NAME as key
  ```
  Example:
  ```
  let x be {'key1': 'value1', 'key2': 'value2'}
  display x as key
  ```
  output:
  ```
  ′key1′,′key2′
  ```

- **pop:**
  Removes the element with the specified key.
  Syntax:
  display VARIABLE_NAME as pop('KEY')
  Example:
  let x be {'key1': 'value1', 'key2': 'value2'}
  display x as pop('key1')
  output:
  'value1'

- **update:**
  Updates the dictionary with the specified key-value pairs.
  Syntax:
  VARIABLE_NAME1 as update(VARIABLE_NAME2)
  Example:
  let x be {'key1': 'value1'}
  let y be {'key2': 'value2'}
  x as update(y)
  display x
  output:
  {'key1': 'value1', 'key2': 'value2'}

- **value:**
  Returns a list of all the values in the dictionary.
  Syntax:
  display VARIABLE_NAME as value
  Example:
  let x be {'key1': 'value1', 'key2': 'value2'}
  display x as value
  output:
  ′value1′,′value2′

v0.0.1 HCPL Code Structure

Conditional Statements

Executes different code based on whether a condition is true or false.

Syntax:

```
if this CONDITION happen then ...
or this CONDITION happen then ...
else this ...
end if
```

Example:

```
let x be 10
let y be 5
if this x > y happen then display formatted string '{x} is greater than 5.'
or this x > 6 and x > 7 happen then display 'x is greater than 6 and 7.'
else this display 'x is smaller than or equal to 5.'
```

output:

```
10 is greater than 5.
```

While Loop

Executes a block of code as long as a condition is true.

Syntax:

```
while CONDITION happen then ...
end while
```

Example:

```
let x be 1
while x < 5 happen then
display x
add 1 to x
end while
```

output:

```
1
2
3
4
```

v0.0.1 HCPL Code Structure

For Loop

Used for iterating over a sequence.

Syntax:

1. for VARIABLE_INITIALIZATION, CONDITION
2. for INITIALIZATION, CONDITION, INCREMENT/DECREMENT

Note: In the first Syntax, the initialized variable is incremented by 1 by default.

Example1:

```
for i be 1, i < 5
display i
```

output:

```
1
2
3
4
```

Example2:

```
for i be 1, i < 5, add 3 to i
display i
```

output:

```
1
4
```

Example3:

```
let fruits be 'apple','banana','cherry'
for i be 0, i < 3
display fruits[i]
```

output:

```
apple
banaba
cherry
```

Example4:

```
let x be 'banana'
for i be 0, i < 6
display x[i]
```

output:

```
b
a
n
a
n
a
```

v0.0.1 HCPL Code Structure

<u>Functions</u>

A block of code that is executed only when it is called.

Syntax:

define function FUNCTION_NAME with PARAMETER(s)

Example1:

```
define function Swagat with x
display 'Hello ' + x
Swagat 'Swagat'
```

output:

```
Hello Swagat
```

Example2:

```
define function parent with *kids
display 'The youngest child is ' + kids
1
parent 'Swagat', 'Manish'
```

output:

```
The youngest child is Manish
```

Example3:

```
define function parent with **kids
display 'The youngest child is ' + kids
young
parent (old = 'Swagat', young = 'Manish')
```

output:

```
The youngest child is Manish
```