Variables

null()
base() - bool(), name(), num()
un()
Augmented Assignment Operators & Concatenators

names(), nums()
Holding, Spreading and Packing by Camouflage & Member

dict() - Layer, tree()
set()

Operators & Functions that work with Terminal
Valueholders

Packing
Renaming
p-any
VarError
avoid & temp Clauses

clear(), del()
copy(), reset(), del_copy()

Comparison Operators - No Operator, =, !=, <, >, <=, >=, is, is not
Variable <it>

String Syntax
Syntax of selfArgument

Main Exhaust & Sharing

Syntax interpretation as we type

Extras
Summary

null()

<name>{none} **or** <name><> **or** <any(name)><> is **null.base()**

<names.list>{none} **or** <names.list><> **or** <names.list>{name} is **null.list()**

*<name>() ~~()~~ <names.list>() generates **SE**;*
*<names.list><'none'> generates **DisE**;*

base()

- It is bit.zero() that can be/ become **member of list** or **value of key**.

  num()
  name()
  bool()

bool()

<n>([])
<statement>(n=n*1) § ~~(true)~~ **Here, you generated a Variable and assigned it a value.**
<statement>{true} § ~~(true)~~ **Now, you tried to change its value.**
<statement>(3+2!=5.0) § ~~(false)~~ **Now, you changed its value.**

Now, any of the following conditions doesn't get executed:

if statement;
if statement is true();
if statement = true;

While any of the following conditions gets executed:

if NOT statement;
*if statement is not true(); ****caution: It gets implemented with any type except true().***
if statement != true;

name()

<name><Sarah>,
<name><13>,
<any(names)><Sarah\, 13> are **str()**

- **Snip is special type of name() that can store simple data as well for further modifications.**
  <current_yr>{date("yy", 2019)} § ~~2019~~ is **Snip**
  <current_yr> ++ < is current year.> § ~~2019 is current year.~~ is str()

num()

<num>(octal("010")) § ~~(8)~~,
<num>(octal("010")+2) § ~~(10)~~,
<num>(hex("FAFA")) **or** <num>{hex("fafa")} § ~~(64250)~~ are **int()**
<rating>(15/2) § ~~(7.5)~~ is **float()**

<num>(-3)
abs(num) **or** <num>(0-num) § ~~(3)~~

*<num><(4)> generates **SE**;*
<num><\(4\)> § ~~(4)~~ is str()

How to generate num():

<num>(4)
<num>{double(2)}
<num>(double(2))

How to copy num():

<another_num>{num}
<another_num>(num)

Note: <another_num><'num'> § 4 is str()

un()

<n>(3) is num()
<num>(n) is num()

<n>([]) is **str.un()** **which is non-modifiable.**
<num>(n) is **num.un()**
<statement>(n=0) is **bool.un()**

<n>([])
<num>(n)
del(n)
<statement>(num+1>num) § ~~(n+1>n)~~
*Here, <statement>(n+1>n) generates **NameE**;*

str.un() as camouflagedVar supports only outOperator:

> *<num>{~~n~~} generates **VarE**;*
> <out>{n} $ ~~(n)~~

num.un() must hold at least one str.un(). Else, it is num():

> (n), (n+0), (n*1) ~ (n),
> (n+n-n), (n*n/n) ~ (n) are num.un()
>
> (n-n) ~ (0),
> (n+2-n), (n*2/n) ~ (2) are num()

Arrangement within num.un() is first num() then str.un() chronologically as they have been defined:

Suppose <n>([]) has already been defined before <m>([]):

> (-n+3) ~ (3-n)
> (m-n**2) ~ (-n**2+m)

Within bool.un():

if main and rfr both are num.un()/ str.un(),

if both are of same value, it is bool(). [=, <=, >= means (true), <, >, != means (false)]

(n+1=1+n) § ~~(true)~~

(n!=n+0) § ~~(false)~~ are bool()

else, it is bool.un(). It transfers num only with +, - operators.

(n+1>5) ~ (n>4)

(5>n+1) ~ (4>n)

(n*3>8), (n/7=1.143), (n//7=1), (n**2>5), (2**n>5) all are bool.un()

(n+1>n-1) ~ (n>n-2) is also bool.un() **(even if it is always true())**

else, bool.un() can also have one num() and one num.un()/ str.un().

(n>5), (5>n) are bool.un()

bool.un() as argument of function:

*<out>{fun(3➢4)} generates **SE**;*

*<out>{fun(3=4, 5)} generates **FIE**; (use of keyword argument must follow the same.)*

*<out>{fun(3=4)} generates **FIE**; (keyword argument can't use any numericals.)*

<n>([])

<out>{fun(n=4)} has no bool.un() as argument.

Note: It works only if fun() accepts keyword argument "n".

Augmented Assignment Operators

- **Works on num(), num.un().**

<str><>

*<~~str~~> += 2 generates **DisE**; (by string syntax.)*

*<~~str~~> *= 2 [<str>(~~str~~*2)] generates **DisE**;*

<n>([])

*<~~n~~> += 2 [<~~n~~>(n+2)] generates **VarE**;*

<price>(0)

<price> += 7 **is** <price>(price+7) § ~~(7)~~

<price> //= 2 **is** <price>(price//2) § ~~(3)~~

<n>([])

<num>(n)

<num> += 2

or

<num>(2)

<num> += n **or** <num> += (n) § ~~(2+n)~~

*Here, <num> += ~~<~~n> generates **SE**;*

<num>(2)

<num> *= 2+n **or** <num> *= (2+n) **is** <num>(num*(2+n)) § ~~(4+2*n)~~ **vs**

<num>(num*2+n) § ~~(4+n)~~ **vs**

<num>(num*n+2) § ~~(2+2*n)~~

<num>(2)

<num> *= (n+3)*5 **or** <num> *= ((n+3)*5) **is** <num>(num*((n+3)*5)) **is** <num>(10*(3+n))

§ ~~(30+10*n)~~

<num>(3)
<nums.list>(1, 2, 3, 4)

<num> -= (.get (1){nums.list}) **or** <num> -= .len{nums.list}-3 § ~~(2)~~

Here, <num> -= .get (1)≼nums.list>
*<num> -= .len≼nums.list>-3 Both generate **SE**;*

Concatenators

- **Works on name(), null.base().**

Syntax for local:

<var> ++ <str>
<var> ++ < >

Syntax if end return is name():

<var> += existing_var
<var> += space
<var> += .property{existing_var}
<var> += fun(existing_var)

Syntax if end return is num(), 1d-list():

<var> ++ <'existing_var'>
<var> ++ <'.property{existing_var}'>
<var> ++ <'fun(existing_var)'>

<greeting><>
<greeting> ++ <Hi\, > works.

<greeting><Hi\, >
<greeting> += none
*<greeting> += true Both generate **DisE**;*

<greeting><Hi\, >
<name><John>
<num>(13)
<names.list><John, Yara>

<greeting> ++ <Jon> **or** <greeting> += .remove h{name}
§ ~~Hi, Jon~~

<greeting> ++ <13> **or** <greeting> ++ <'num'> **or** <greeting> += stringify(num)
§ ~~Hi, 13~~

<greeting> ++ <John\, Yara> **or** <greeting> ++ <'names.list'> **or** <greeting> += stringify(names.list)
§ ~~Hi, John, Yara~~

<greeting> ++ <My name is 'name'.> **has combination.**
§ ~~Hi, My name is John.~~

<age><I am >
<age> += dt.design(date("y", 28), "%ue")
<age> ++ < old.>
§ ~~I am 28 years old.~~

names()

- **Member can be base(), null.list(), 1d-nums(), 1d-names().**

<any(availabilities)><‘true’, ‘false’> § ~~<(true), (false)>~~ is **fbool().**

<leap_years.list><‘date(“yy”, 2000)’, ‘date(“yy”, 2004)’> § ~~<2000, 2004>~~ is **fsnip().**

<lst.list>(()) § ~~<<>>~~ is **fnull().**
Here, <lst.list>((), ()) is also fnull().
Note: Here, <lst.list><<>> also works. *But* ***<lst.list><<()>>*** *generates* ***SE;***

<lst.list><**(0)**, John> § ~~<(0), John>~~ Here, first member is num().
For first member using <num>(1),
(num), **(num-1)**, **‘num’** also works. *But,* ***‘0’, ‘num-1’*** *generates* ***SE;***

<chars.list>{of_str(“My name”, 1)} § ~~<M, y, , n, a, m, e>~~ Here, third member is <space>.

<any(availabilities)>(3>1, 3!=3)
<any(availabilities)><(3>1), (3!=3)> Both generates SE; (p-list doesn’t allow use of Comparison Operator within round bracket.) ***

<names.list><John, ‘space’, Sarah> works.
<names.list><John, , Sarah> generates ***SE;***
<stocks.list><yellow: ‘true’, red, black> generates ***SE;***

<lst.list><Sarah, (13), <>, <John>, <(13)>>
Here, member types in order are: str(), int(), null.list(), names(), nums().

.get (3) .append (13)<lst.list>
.get (4) .insert (1), (13)<lst.list>
.get (5) .append John<lst.list>
§ ~~<Sarah, (13), <(13)>, <(13), John>, <(13), John>>~~
Now order is: str(), int(), nums(), names(), names().

nums()

- **Member can be num(), null.list(), 1d-nums().**

<num>(1)
<nums.list>(num:3)

or <nums.list>(num, {double(2)-3)*2, double(2)-1)
Note: Here, "{" is math bracket.

or <nums.list><(num), ((double(2)-3)*2), {double(2)-1)>
Note: Here, "{" is num bracket.
Without it, member double{2)-1 generates SE;

§ ~~<(1, 2, 3)>~~

<nums.list>(num, 2, ())
Note: *<nums.list>(num, 2, <>) generates* ***SE;*** ***(p-list doesn't allow use of Comparison Operator within round bracket.)***

or <nums.list><(num), (2), <>>
Note: *<nums.list><(num), (2), (}> generates* ***SE****; (It should be type num().)*
Or it should be string as \(\).

§ ~~<(1, 2, ())>~~ **Here, third member is null.list().**

<nums.list><'num'., (2), (3)>
§ ~~<1., (2), (3)>~~ **is not nums(), but names().**

Holding, Spreading and Packing of the value

Camouflage can spread values across members.
Range as a whole can spread numbers across members.

```
def one_to_three():
        get none;
        return (1:3);
```

<temp.list><John, Yara, Sarah>
<names.list>{temp.list} § ~~<John, Yara, Sarah>~~ has length (3).
<nums.list>{one_to_three()}, <nums.list>(1:3) § ~~<(1, 2, 3)>~~ has length (3).
<nums.list>(1:1) § ~~<(1)>~~ spreads data to only one member.

Member of typed-in angle bracket can hold list() within single quote but Member of typed-in round bracket can not hold nums(), null.list() or even range() without another set of round bracket.

It can't spread it to nearby members.

<any(first_couple)><Sarah, Mosh>
<couples.list><'first_couple', <John, Yara>, Mia>
§ ~~<<Sarah, Mosh>, <John, Yara>, Mia>~~ has length (3).

<nums.list><'one_to_three()'> **or**
<nums.list>((one_to_three()))
§ ~~<((1, 2, 3))>~~ has length (1).

*<nums.list>(one_to_three()) generates **DisE**; (member of typed-in round bracket **needs another set of round bracket** to dissolve nums(), null.list().)*
***Note:** <nums.list><<'one_to_three()'>> generates **DisE**; (It can dissolve **only base().**)*

<null.list><>
<nums.list><‘one_to_three()’, ‘double(2)’, <(5:7)>, ‘null.list’> **or**
<nums.list>((one_to_three()), double(2), (5:7), (null.list))
§ ~~<((1, 2, 3), 4, (5, 6, 7), ())>~~ has length (4).

*<nums.list>(one_to_three()), <nums.list>(null.list) generate **DisE**;*

<nums.list>(5:7) works.
*<nums.list>(4, 5:7), <nums.list>(5:7, 8) generate **SE**;*

Camouflage and member can do the packing, but unlike Camouflage, member can’t spread it to nearby members.

```
def one_to_three():
        get none;
        return 1, 2, 3;

<nums.list>{one_to_three()}
§ ~~<(1, 2, 3)>~~ has length (3).
<nums.list><‘one_to_three()’>, <nums.list>((one_to_three()))
§ ~~<((1, 2, 3))>~~ has length (1).
```

dict()

- **Member must be one or more key-value pairs.**

Key

Can be num().
(0): (1): **or** (1.0): (1.5): (1+1):

Can be str()
none: true:
*'none': generates **DisE**; (*** see tree()**) (true): generates **DisE**;*

*(space): generates **DisE**;*
'space': **or** \ :
    <any(file)><The\ Office: (22) Friends: (17)>
    <any(chars)><M: <(1)> y: <(2)> \ : <(3, 8, 11, 17, 19, 22)> n: <(4, 15)>>
space: is different key than above.

*(1:3): generates **SE**;*
    <dct.list><\(1\:3\): one to three> has string "(1:3)" as key.
    ..fetch one to three<out>{dct.list} $ ~~(1:3)~~

*'date("y", 29)': generates **DisE**;*

*<n>([]) (n): generates **DisE**;*
<n>(3) (n): works.

ckey

It must be str() that contains only alphabets [a-z, A-Z].
It doesn't allow space or any numerical [0-9].

ckey: name: Available:

Value

Can’t be null.base(), local bool(), un(), tree().
Note: tree() of dict() is not possible.

It can be following:

name: John
detail: My name is John.

*<n>([]) num: (n) generates **DisE;***
<n>(3) num: (n) works.

available: ‘true’
*available: (true) generates **DisE;***
*available: (3>1) generates **SE;** **(p-list doesn’t allow use of Comparison Operator within round bracket.)***

lst: <>
couples: <<Sarah, Mosh>, <John, Yara>, Mia>
nums: <(1:3)> **or** nums: <(1, 2, 3)>
2d: <((1, 2, 3))>
*nums: (1; 2, 3) 2d: ((1; 2, 3)) generate **SE;***

name: <ckey: John>
num_to_word: <(1) : One, Two : (2)>

Defined value

Can’t be null.list(), 2d-list(), dict(), set() as well.

Inner value (Value of value-dict())

Can’t be 2d-list(), dict(), set() as well.

Layer

If different member's "ckey" has same typed-in value, Layer combines these members.

<ckey: John wife: Yara,
ckey: John daughter: Sarah>

§ ~~<ckey: John wife: Yara daughter: Sarah>~~

This property of Layer doesn't allow tree() to be Layer.

If its typed-in value is 1d-list(), it represents different members with same key-value pairs.

<ckey: <John, Yara> daughter: Sarah>

§ ~~<ckey: John daughter: Sarah,~~
~~ckey: Yara daughter: Sarah>~~

1d-list as its print value is made by alt_layers().

<ckey: John wife: Yara>
        // alt_layers("John", "Jon")

§ ~~<ckey: <John, Jon> wife: Yara>~~

<ckey: John wife: Yara>
        // alt_layers("John", none)

§ ~~<ckey: <John> wife: Yara>~~

tree()

It is non-modifiable.
It doesn't allow typed-in "linked" key.
Note: "linked" can be used with .get. as continuous selfArgument. It can't be used with .get,pair.

It doesn't allow typed-in 2d-list(), dict(), set().

- **Value of "linked" key:**
  null.list()
  Non-defined dict(**=1**) with key's value being base(), 1d-nums(), 1d-names()

- **Value of other linked keys if they are available:**
  Defined dict(of any length) with ckey same as that of tree()

(e5)
<dct.list>{family_tree.list} is also tree()

..get (1:-1)<dct.list>{family_tree.list} **or**
<dct.list><>
.append,nis<dct.list>{family_tree.list}
Here, <dct.list> is not tree() (has no "linked" key). It also has no linked key present.

ofTree(family_tree.list : dct.list) **(of easyA env.)**
It is not tree() but it has all linked keys present.

ofTree(family_tree.list, 1: member.list)
Here, <member.list> has linked keys present.

: for member.list in family_tree.list **or**
: for member.list in .get (1:3){family_tree.list}
Here, <member.list> has no "linked" or linked keys present.

: for member.list in tree(family_tree.list) **or**
: for member.list in tree(family_tree.list, 1, 3)
Here, <member.list> has linked keys present.

Concept of opposite key

Main and opposite both keys should be str() [with only alphabets]
*(1)-(1.0): generates **SE**;* **Note:** (1-1.0): **is same as** (0):

You can use it only <u>in dependent part of tree</u>().
<detail.list><name: John : wife-husband: Yara,
name: Yara>

Here, John's wife —> Yara
Yara's husband —> John

*** Concept of <none> as key**

In dependent part of tree(), if used as main key, it means current member has no relation to upcoming member. But if used as opposite key, it means upcoming member has no relation to current member.

<detail.list><name: John : wife-'none': Yara,
name: Yara>

Here, John's wife —> Yara only
*So, ..find Yara ..get husband<out>{detail.list} generates **AAE**;*

<detail.list><name: John : 'none'-husband: Yara,
name: Yara>

Here, Yara's husband —> John only
*So, ..get (1), wife<out>{detail.list} generates **AAE**;*

set()

- **Member can be single set with num() or name() as a value.**

<num_eng.list><(3) : Three,
(33) : Thirty three> **is set of int() vs str().**

<rating.list><(9+0.8) : good,
(9.9) : best> **is set of float() vs str().**

<months.list><'date("mm", 1)' : January,
'date("mm", 2)' : February> **is set of Snip vs str().**

<months.list><March : Mar.,
April : Apr.,
May : May> is set of str() vs str(). (Same value but within same set. So, it works.*)**

<num_eng.list><(1) : One,
*~~(1)~~ : Only> generates **SE**; (value already exists)*

<num>(1)
<num_eng.list><(1) : One,
*(num) : Only> generates **DisE**; (value already exists)*

<num_eng.list><(1) : One,
*~~(1.0)~~ : One point zero> generates **SE**; (set is of int() vs str())*

<doubles.list><(5) : (10),
*~~(10)~~ : (20)> generates **SE**; (value already exists)*

<half_as_float.list><**(10)** : (5.0),
(20) : **(10.0)> is set of int() vs float(). (Same value but type indifference. So, it works.***)**

Restricted set() explained using Fibonacci Sequence:

<fib_cache.list><**(1)** : (0),
(2) : **(1)**> **is set of int() vs int(). (Same value but Restricted set. So, it works.***)**
// is_restricted()

At first position, Fibonacci number is (0), while at second, it is (1).
Here, you can't add (1) or (2) in this set. But .add (0), **(..)**. works.

```
def fib():
    get global(fib_cache.list);
    get pos if it is int() AND it > 0;
    get none;
    if .fetch (pos) .bool{fib_cache.list}:
        return it;
    else:
        fib(pos-1 : previous1)
        fib(pos-2 : previous2)
        <value>(previous1+previous2)
        .add (pos), (value)<fib_cache.list>
        return value;

fib(3 : none)
fib(4 : none)
fib(6 : none)
<out>{fib_cache.list}
$ <(1) : (0),
(2) : (1),
(3) : (1),
(4) : (2),
(5) : (3),
(6) : (5)>
```

*.fetch (0)<out>{fib_cache.list} generates **AAE;***
Boolean bundle .fetch (0) .bool{fib_cache.list} works as false().

.fetch (1)<out>{fib_cache.list} $ (0) **means at first position, Fibonacci number is (0).**

.fetch,all (1)<out>{fib_cache.list} $ ~~<(2, 3)>~~ **means second and third positions have Fibonacci number as (1).**

.fetch. checks main side only while .fetch,all. checks restricted side only.
Note that .fetch,all. generates VPME with Simple set.

Operators

<in>

Use as simpleVar if you don't need what user types on terminal.

<in>What is your name? <>
<out><Hi\, welcome.>

Else, use as camouflagedVar.

Mostly, inOperator makes it str().

<name>{in}What is your name? {} $ ~~13~~
<out>{type(name)} $ ~~(str)~~

Here, mainVar must become p-none. It must be without any property.

*<~~var.list~~>{in}{} generates **SE**;*

<any(var)><John, Yara>
*<var>{in}What is your name? {} generates **VarE**;*

<any(var)><John, Yara>
<any(var)>{in}What is your name? {} works. **Here, <var> becomes p-none.**

*.~~append~~<var>{in}What is your name? {} generates **SE**;*

<out>

With null(), as mainVar

<out><> ~ <out>{none} $ ~~(a line generated by /b(null))~~
<temp.list><> <out>{temp.list} $ ~~<>~~

With bool(), as mainVar

<out>{true} $ ~~(true)~~
<statement>(2>3) <out>{statement} $ ~~(false)~~ ***Note:*** *<out>(2~~>~~3) generates **SE**;*

With un(), as mainVar

<n>([]) <out>{n} $ ~~(n)~~
<num>(n+1) <out>{num} $ ~~(1+n)~~
<statement>(n>1) <out>{statement} $ ~~(n>1)~~

With num(), str(), also as simpleVar

<out>(3+4) $ ~~(7)~~ is as simpleVar.
<out>< > $– is as simpleVar. Can also be as ~ <out>{space}.
<name><John> <out><Hello\, 'name'.> $ ~~Hello, John.~~ is as simpleVar.

..remove h<out>{name} $ ~~Jon~~ Here, camouflagedVar has previousDot property.

<out>{date("y", 29)} $ ~~29~~

With list(), dict(), set(), Function, as mainVar

<nums.list>(1:3) <out>{nums.list} $ ~~<(1, 2, 3)>~~ ***Note:*** *<out>(1:3) generates **SE**;*
<availability.list><'true'> <out>{availability.list} $ ~~<(true)>~~ ***Note:*** *<out><'true'> generates **DisE**; (by String Syntax)*

<out>{double(2)} $ ~~(4)~~

Interpretation of what outOperator prints on terminal:

<num>(1) <out>{num} $ ~~(1)~~
<out>{true} $ ~~(true)~~
<n>([]) <out>{n} $ ~~(n)~~

- **Round bracket** holds num(), bool(), un().

<name><John> <out>{name} $ ~~John~~
<out>{date("m", 9)} $ ~~09~~

- **No bracket** means it is name().

<nums.list>(1) <out>{nums.list} $ ~~<(1)>~~
<names.list><John> <out>{names.list} $ ~~<John>~~
<name.list><name: John> <out>{name.list} $ ~~<name: John>~~
<set.list><(1) : One> <out>{set.list} $ ~~<(1) : One>~~

- Angle bracket enclosing round bracket **<(...)>** means it is nums().
  While angle bracket only means it is names(), dict(), set().

- If nums() has round bracket, that member is null.list() or nums().
  If names() has round bracket, that member is num() or bool().
  If names() has angle bracket, that member is list().

<(1, (), 3)> is 2d-nums().
<(1), (true), (3)> is 1d-names().
<((1))> is 2d-nums().
<<(true)>> is 2d-names().

Functions that work with Terminal:

loop:
: <num>{in} Guess a number between 1 and 99\: {}
try:
beint(num : num) **“zero” makes try clause unsuccessful instead of generating FIE and collects after clause.**
if in_range(num, 1, 99): **With (0), try clause is successful and it collects after clause.**
collect(“f”) **With (1), try clause is successful and it collects finally clause.**
after:
<out><‘num’ is not a valid option.>

or loop:
: <num>{in} Guess a number between 1 and 99\: {}
if bool(beint(num)) AND in_range(num, 1, 99):
<num>{it} del(it)
collect(“f”)
<out><‘num’ is not a valid option.>

or loop:
: <num>{in} Guess a number between 1 and 99\: {}
// t.num() **“zero” isn’t allowed at terminal itself.**
“001” ~ (1) | “.3” ~ (0.3) | “-.3” ~ (-0.3) | “0.250” ~ (0.25)
after:
<out><‘num’ is not a valid option.>
until:
if num is int() AND in_range(num, 1, 99);

loop:
: <rating>{in} Rate this movie\: {}
// t.num()
then <rating>{in} {}
// t.num()
: if in_range(rating, 1, 10):
collect(“f”)

```
finally:
        float(rating)



<statement>{in} Are you existing user? {}
        // t.bool()

if statement:
        <out><Welcome back...>



<password>{in} Enter your password\: {}
        // t.len(8, 15) Here, first input is >0 while second input is >= first input.
        t.secure() is t.secure("*")



<greeting><Hi\, John. >
<message>{in} Type a message\: {} Here, message is always str().
<greeting> += message
<out>{greeting}
$ Type a message: enter (which won't work) "welcome"enter
$ Hi, John. welcome
$

<greeting><Hi\, John. >
<message>{in} Type a message\: {} Here, message can be null.base().
        // t.null()
if message is not null():
        <greeting> += message
<out>{greeting}
$ Type a message: enter (which works)
$ Hi, John. |
$
```

Valueholders

<none> is null.base().
<space> is str().
<true>, <false> are bool().

- *Generates **SE** as simpleVar, mainVar:*
  <~~none~~><> (used as mV)
  <name><John> .append<~~none~~>{name} (used as mV)
  <~~true~~><yes> (used as sV)
  <~~space~~> ++ <John> (used as sV)

<temp>{none} is null.base() while <temp.list>{none} is null.list() (used as cV)

*.append<temp.list>{temp} generates **ADE**;*
<name><John> ..remove John<temp.list>{name} § ~~<>~~
*..remove John .append<temp.list>{name} generates **ADE**;*

*.append (~~none~~)<temp.list> generates **SE**; (used as dV)*
*.append (~~.remove John{name}~~)<temp.list> generates **DisE**;*
.append (temp.list)<temp.list> § ~~<<>>~~

<var.list>{true} **or** <var.list><> .append<var.list>{true} (used as cV)
or <var.list><> .append (true)<var.list> § ~~<(true)>~~ (used as dV)

Now, <var.list>{false} becomes § ~~<(false)>~~,
While .append<var.list>{false} becomes § ~~<(true), (false)>~~.

<items.list><available: 'true'> § ~~<available: (true)>~~ (used as dV)

<detail>{space} (used as cV)
or <detail><'space'> (used as dV)
or <detail><> <detail> += space (used as dV)

Packing

<num>(3)
<nums.list>{num} § ~~<(3)>~~
*Note: <num>{nums.list} generates **VarE**;*

<availability.list>{true} § ~~<(true)>~~

Note: <lst.list>{none} is not packing but is <lst.list><>

Camouflage, Member of list, selfArgument, Input of function, transferDependent position (function's return statement) can do the packing.

Camouflage can also spread it to nearby members.

def one_to_three():
    get none;
    return 1, 2, 3; **has three return statements.**

<nums.list>{one_to_three()} § ~~<(1, 2, 3)>~~

<nums.list><'one_to_three()'> **or** <nums.list>((one_to_three())) § ~~<((1, 2, 3))>~~

<nums.list><>
.append (one_to_three())<nums.list> § ~~<((1, 2, 3))>~~

*num_sum(one_to_three() : sum) generates **FIE**; (by num_sum().)*
nums_sum(one_to_three() : sum) § ~~(6)~~

def no_purpose():
    get none;
    return one_to_three(); **has only one return statement.**

Renaming

<name>{~~in~~} What is your name? {} del(in)
<~~out~~>{name} del(name) Both generate **SE**;

For **(e8)**, ..*~~get~~ apple<apples.list>{fruits.list} del(fruits.list) generates **SE**;*
For **(e5)**, <any(details)>{family_tree.list} del(family_tree.list) Here, details is tree();

For <name><John>, <any(name)><John>,
    <any(var)>{name} del(name), <var>{name} del(name) works.
    Now, <out>{var} $ ~~John~~ and *<out>{name} generates **NameE**;*

    *<var.list>{name} del(name) generates **VarE**;*

For <any(name)><John, Yara>,
    <any(var)>{name} del(name), <var.list>{name} del(name) works.

    *<var>{name} del(name) generates **VarE**;*

For <names.list><John>,

    <any(var)>{names.list} del(names.list), <var.list>{names.list} del(names.list) works.

    *<var>{names.list} del(names.list) generate **VarE**;*

<any(name)><John, Yara>
<names.list><John>
<name>{names.list} del(names.list)
<out>{name} $ ~~<John>~~

p-any

<any(names)><John, Yara> **It creates p-list.**
*<out>{names.list} generates **NameE**;*
Here, Variable <names> exists but <names.list> doesn't.

<name><John>
<name.list><name: 'name'>
<any(name)>{name.list} **It changes p-none to p-list.**
*Note: <name>{name.list} generates **VarE**;*
<out>{name} $ ~~<name: John>~~
<out>{name.list} $ ~~<name: John>~~

get var; can receive p-none.
get var.list; can receive p-list.
get any(var); can receive p-none or p-list.

<var><John> § ~~John~~ is str().

How to change its pointer to p-list.

> *Here, clear(var) doesn't work, as it makes it null.base()*.
>
> <any(var)><John, Yara>
> .pop<var>
>
> <temp.list>{var} is **packing**.
> <any(var)>{temp.list} del(temp.list) is **renaming**.
>
> **pack(var : any(var))**
> *Here, pack(var : var) generates **FOE**;*

<out>{var} $ ~~<John>~~ is list().
<var><name: John> § ~~<name: John>~~ is now one().
<var><(1) : One> § ~~<(1) : One>~~ is now set().

Now, how to change its pointer back to p-none.

Here, clear(var) doesn't work, as it makes it null.list().

reset(var)
Note: It works only if you have copied Variable by copy().
*It generates **FIE** if you have deleted Variable in-between, as it can't access its copy.*

del(var)
<var><John>

<any(var)><John>
Here, <var><John> is still list().

..get (1)<temp>{var}
<any(var)>{temp}
Here, <var>{temp} is packing. So Variable is still list().

unpack(var : any(var))
*Here, unpack(var : var) generates **FOE**;*

..get (1)<any(var)>{var}
Here, ..get (1)<var>{var} is packing. So Variable is still list().

.get (1)<any(var)>
Here, .get (1)<var> is packing. So Variable is still list().

<out>{var} $ ~~John~~

How to generate list(=1) using p-any.

<name><John>
<num>(5)

Here, <any(names)><John> ***or*** *<any(names)>{name}* ***or*** *<any(names)><'name'> is str().*
<any(nums)>(5) ***or*** *<any(nums)>{num}* ***or*** *<any(nums)>(num) is num().*
<any(nums)><'num'> is str().

<any(names)><'name', (0)>
.pop<names>
<any(nums)><'num', (0)> **or** <any(nums)>(num, 0)
.pop<nums>

How to copy the list.

<nums.list>(1:3)

As it is.
<example.list>{nums.list} <out>{example.list} **or**
<out>{nums.list} **or**
<any(example)>{nums.list} <out>{example}
$ ~~<(1, 2, 3)>~~

As a string.
<out><'nums.list'>*** or**
<any(example)><'nums.list'>****** <out>{example}
$ ~~1, 2, 3~~
*** outOperator as simpleVar is p-none. So, it is String Syntax.**
**** p-any chooses p-none first.**

As a two-dimensional list.
<example.list><'nums.list'> <out>{example.list} **or**
<any(example)><'nums.list', (0)> .pop<example> <out>{example}
$ ~~<((1, 2, 3))>~~

VarError

<names.list><>
<any(names)>{names.list}
*Here, <~~names~~>{names.list} generates **VarE**;*

*if names.~~list~~ = none; generates **SE**;*
*if ~~names~~ = none; generates **VarE**; (type not supported.)*

<var>{in}What is your name? {}
*Here, <~~var.list~~>{in}{} generates **SE**;*

<any(var)><John, Yara>
*<~~var~~>{in}What is your name? {} generates **VarE**;*

<n>([])
*Here, <~~n~~>(2), <~~n~~> += 2 both generate **VarE**;*
<num>(n)
*Here, <num>{~~n~~}, <any(num)>{~~n~~}, <num.list>{~~n~~} all generate **VarE**; **(str.un() as camouflagedVar supports only outOperator)***

<var>{num}, <any(var)>{num} both work.
*<var.list>{~~num~~} generates **VarE**;*
*<var.list><'num'> generates **DisE**;*

For <var.list><>,
*.append<var.list>{~~n~~} generates **VarE**;*
*.append<var.list>{num} generates **ADE**;*

*if ~~n~~ = ~~n~~ generates **VarE**; (type not supported.)*
*if ~~num~~ > ~~n~~; generates **VarE**; (type not supported.)*

avoid Clause & Block

To prevent errors:

(e31)

avoid AFE:
.add (33), Thirty three<set.list>

avoid ADE with .add (32), (32.0)<set.list>

To generate error ADE:

- **Properties, Variable receiving segment, ";" Operator seek permission from main env. when type-change is necessary. avoid clause/ block with typeChange option denies that permission.**

<nums.list>(1, 2, 3)
<names.list><John, Yara, Sarah>

avoid typeChange: **It raises ADE on type-change.**
avoid ADE with .append John<nums.list>;
avoid ADE with .append<nums.list>{none}; **Note that it doesn't ask for type-change permission.**
avoid ADE with .insert (1), (0)<names.list>;

<out>{nums.list} $ ~~<(1, 2, 3)>~~
<out>{names.list} $ ~~<(0), John, Yara, Sarah>~~

<availability.list><'true'>
// is_smpl()

avoid typeChange with .append (0)<availability.list>;
***and** avoid typeChange with .pop<availability.list>; generate **ADE**;*

<couples.list><<(0, 1)>, <John, Yara>>

.get (1)<couples.list> makes it nums().
***so** avoid typeChange with .get (1)<couples.list>; generates **ADE**;*

.get (2), (1)<couples.list> is packing of valueResult ~~"John"~~ which doesn't change type.
so avoid typeChange with .get (2), (1)<couples.list>; works.

.get (2), (1); .remove h<couples.list> is also packing of endResult ~~"Jon"~~ which doesn't change type. But, there is temporary type-change with ; operator which makes valueResult ~~"John"~~ into string.
***so** avoid typeChange with .get (2), (1); .remove h<couples.list>; generates **ADE**;*

(e8)

avoid typeChange:
*.get apple<fruits.list> generates **ADE**; (one() to many() is type-change.)*

avoid typeChange:
.get apple, (1)<fruits.list> works. (Simple-dict to Layer isn't type-change.)

It works with properties only:

<name><>
<names.list><John, Yara>

avoid typeChange:
<name><John>
<out>{name} $ ~~John~~ works.

avoid typeChange:
ofLst(names.list, "name" : names.list) **of easyA env.**
<out>{names.list} $ ~~<name: John, name: Yara>~~ works.

temp Clause

<nums.list>(1, 3, 5, 7)
<num>(3)

temp:
..get (-1)<another_num>{nums.list} **It gets generated.**
.pop<nums.list> **It gets updated but since fill() returns it unconditionally, temp clause doesn't copy it.**
fill(nums.list, another_num) **means fill main env. with these two variables.**

temp:
<num> += another_num **temp clause doesn't make a copy.**
if num >= 10:
<num>{another_num} del(another_num)
fill(num)

- **Note that main env. supports deletion, renaming done within temp clause.**
  Also note that variable cleared within temp clause would be effective in main env. only with fill().

temp:
<num> += 1 **temp clause makes a copy.**
if num/2 is int():
<out>{num} $ ~~(8)~~

<out>{num} $ ~~(7)~~

clear(), del()

Can be used only at simplePosition. [Note: Renaming also uses del() at simplePosition!]

Both take one argument only.
*Here, local argument generates **FIE**;*

clear() makes bit.zero() as null.base() and bit.one() as null.list():

> <any(example)><John> is str()
> clear(example) makes it null.base().
> *.append Yara<example> generates **VPME**;*
> **Note:** *<example>{clear()} generates **FE**;*
>
> <any(example)><John, Yara> is names()
> clear(example) makes it null.list().
> .add name, John<example> works.

del() can also delete non-modifiable types:

> <n>([])
> *<n>(4) **or** double(2 : n) generates **VarE**;*
> *clear(n) generates **FIE**;*
>
> del(n)
> <n>(4) works.
>
> <n>(4)
> <n>([]) works without del() as type num() is modifiable.

Use of del() within function:

def fun():

get names.list; ***or*** *get rfr(names.list);* ***or*** *get names.list as self;*

del(names.list) ***generates FSE; (DenyE by del())***

def fun():

get names.list++;

del(names.list) **works only within function.**

def fun():

get global(names.list);

del(names.list) ***generates FSE; (DenyE by del())***

def fun():

get global(names.list);

del global(names.list); **works also in main env.**

<names.list><John, Yara>

fun()

Now, <out>{names.list} generates ***NameE****;*

Note:

def fun():

get global(names.list);

clear(names.list) **works also in main env.**

copy(), reset(), del_copy()

Can be used only at simplePosition.

Takes one argument.
*Here, local argument generates **FIE**;*

copy() generates FIE with non-modifiable Variables.

del() (as delete or rename) also deletes copy made by copy():

<couple.list><John, Yara>
copy(couple.list)

.append Sara<couple.list>
reset(couple.list)

.append Sarah<couple.list>
<john.list>{couple.list} del(couple.list) **also deletes copy.**

*reset(john.list) generates **FIE**;*
*reset(~~couple.list~~) generates **NameE**;*

reset() generates FIE if it can't access copy:

- *<num>(1)*

  loop:
  *; copy(num) **~ <temp_num>{num} which is local to loop only.***
  : <num> += 3
  until:
  if num >= 10;

  *reset(num) generates **FIE**;*

- <num>(1)

  loop:
  : copy(num) **~ <temp_num>{num} which is given to main env.**
  then <num> += 3
  until:
  if num >= 10;

  reset(num) **~ <num>{temp_num}**
  <out>{num} $ ~~(1)~~

- <num>(1)
  copy(num) **~ <temp_num>{num} which is accessible by loop.**

  loop:
  : <num> += 3
  until:
  if num >= 10:
  reset(num)

  <out>{num} $ ~~(1)~~

- <num>(3)
  copy(num)

  def fun():
  get global(num);
  get none;
  <num> *= 3
  reset(num)

  fun()
  <out>{num} $ ~~(3)~~

- *<num>(3)*
  copy(num)

  def fun():
    get num, none;
    *<num> *= 3*
    reset(num)
    return num;

  *<out>{fun(num)} generates **FSE**; (FIE by reset())*

reset() can also be used on position generated by var is of:

<nums.list>(2, 4, 8, 16, 32)

loop:
  ; ..len<len>{nums.list}
  ; <temp.list><>

  : n is of nums.list AND N is of len
  : if n < 10:
    .append (n)<temp.list>
    <N> += 1
  else:
    reset(N)
until:
  if .len{temp.list} = len:
    <out>{temp.list}

$ ~~<(2, 4, 8, 2, 4)>~~

Comparison Operators

No Comparison Operator

- Supports
  - type - bool()
  - valueholders - <true>, <false>
  - operator - NOT
- It doesn't support local bool().

<statement1>(3 < 1)
<n>([])
<statement2>(n != n)

if NOT statement1;
if NOT statement2;

if ~~NOT~~ 3 < 1;
*if ~~NOT~~ n != n; both generate **SE**;*

if 3 > 1; actually uses ">" operator.
*if ~~n~~ = ~~n~~; generates **VarE**; (type not supported.)*

<names.list><John, Yara>
if NOT .find Mia .bool{names.list};

Use of Comparison Operator

- Here, main side doesn't support Valueholders.

if ~~none~~ = "J";
if ~~none~~ is null.base();
if ~~none~~ is not null.list();
if ~~true~~ != false;
*if ~~space~~ is as(" "); all generates **SE**;*

1) =, !=

- Supports
  - types - null.base(), num(), name(), bool()
  - valueholders - <true>, <false>, <none>, <space>
  - operator - and, or

- Here, int() and float() are same.

- Locally, num() is supported by both sides.
  Quoted string is supported by only comparison side.

<name><>
<names.list><>
<any(names)>{names.list}
    if name != “John”; **(type-indifference)**
    if name != 3; **(type-indifference)**
    if name != true; **(type-indifference)**
    if name = none;
    *if names.~~list~~ = none; generates **SE**;*
    *if ~~names~~ = none; generates **VarE**; (type not supported.)*

<num>(4.0)
<null.list><>
    if double(3)-2 = num;
    if num != .len{null.list};

<name><John>
<lst.list><(0), Jon>
<char.list><J>
<any(chars)>{char.list}

    if name != .get (2){lst.list};
    if .remove h{name} = “Jon”;
    if name != .get (1){lst.list}; **(type-indifference)**

if name or .get (1){char.list} = “J”;
if name or .get (1){char.list} != “J”;
if 3-3.0 and .get (1){lst.list} = 0;
if name and 3 != 0; **(also has type-indifference)**

if .get (1){char.list} = “M” or “J”;
if .get (1){char.list} != “M” or “J”;
if .get (1){char.list} != “M” and “J”;
if .get (1){char.list} = “M” and “J”; **(never gets implemented) *****

*if char.list is name() AND char.~~list~~ = “J”; generates **SE**;*
if chars is name() AND chars = “J”; doesn’t get implemented, doesn’t show any error. ***

Note: AND, OR can be used to combine two statements.

<statement>(1+2 >= 3.0)

if statement != 4; **(type-indifference)**
if statement != “true”; **(type-indifference)**
if statement != false;
if statement = true;
*if 1+2 >= 3.0-= true; generates **SE**;*
Here, if 1+2 >= 3.0 actually uses “>=” operator.

2) <, >, <=, >=

- Supports
  - type - num()
- Main side supports
  - operator - and, or

- Local num() is supported by both sides.
  if -1 < 0;

<num>(13)
    if num <= 15;

<n>([])
<num>(n+1)
    *if ~~num~~ > ~~n~~; generates **VarE**; (type not supported.)*

import datetime
<age>{date("y", 13)}
    if age is num() AND age < 16; doesn't get implemented, doesn't show any error.

3) is, is not

- Supports
  - types - all
  - operators - and, or

- Comparison side must be comparison function.
- Local num() is also supported by main side.

<name>{none}
<names.list>{none}
    if name is null.base();
    if names.list is null.list();
    if name and names.list is null();

if num = 8; gets implemented for <num>(8), <num>(8.0)
if num is int() AND num = 8; **gets implemented for <num>(8) only**

<age>{date("y", 13)}
    if age is int() AND in_range(age, 15, 18); doesn't get implemented, doesn't show any error.

if in_range(rating, 9, 10) AND rating != 9 and 10; gets implemented for <rating>(9.9), doesn't for <rating>(9), <rating>(10.0)

```
if 4.0 is float() and float(0);
if 4.1 is float() AND 4.1 is not float(0);

<name><Jon>
<names.list><John>
        if name and .get (1){names.list} is str() and of(“J”);

<detail><My name is John.>
        if .find John .bool{detail} is true() AND .find Yara .bool{detail} = false:
                <detail> ++ < Yara is my wife.>

<n>([])
<num>(n+1)
<statement>(num>n)  ~>  <statement>(n+1>n)
        if n is str.un();
        if num is num.un();
        if statement is not bool();
        if statement is not true();
        and if statement is not false(); both statements get implemented.

<names.list><John, Yara>
<nums.list>(1)

        if names.list or nums.list is list():
                <out><I have at least one list.>

        if names.list and nums.list is list():
                join(names.list, nums.list : joined.list)

        if names.list is not null.list() and nums() AND NOT .find Mia .bool{names.list};

<details.list><name: John, name: Yara>

        if details.list is names() or many():
                loop:
                        : for any(member) in detail.list
                        : <out>{member}
```

How to check for dict()

- Suppose <any(var)> is Dictionary,

  if var is one(); is for **dict(=1)**
  if var is many(); is for **dict(>1)**

  if var is dict() AND ckey(var) is not null(); is for **Defined dict**
  if var is dict() AND ckey(var) = “ckey”; is for **Layer**

  if var is tree(); is for **tree()** which is always dict(>1)

How to compare snip()

With =, != operators, even if there is snip on both sides, it checks only str data of snip.

- To compare two snips, snips(snip1, snip2)

<age>{date(“y”, 9)}

  if date(“m”, 9) = age;
  if date.design(age, “%au”) != age;

  if NOT snips(age, date(“m”, 9));
  if snips(age, date.design(age, “%au”))

  if [age is num() AND age = 9] OR [age is str() AND age = “09”] OR
  [age is name() AND snips(age, date(“m”, 9))]; doesn’t get implemented, doesn’t show any error. (age is not num(), it is not str(), snips() function returns false) *

How to compare bit.one()

- To compare two 2d-list() (**can be 1d-list() also**), lsts(lst1.list, lst2.list)
- For two dict() (**can be of any length**), dcts(dct1.list, dct2.list)
- For two set() sets(set1.list, set2.list)

<name.list><name: John>
<defined.list><name: John>
	// pack_ckey(“name”)
<layer.list><ckey: John>

	if dcts(name.list, defined.list)
	if NOT dcts(defined.list, layer.list)

How to differ bool()

Suppose statement is bool():
	if statement: **..**
	else: **..**
Suppose statement is bool() or bool.un():
	if statement is true(): **..**
	elif statement is false(): **..**
	# else, Error
Suppose statement is base() or null.base():
	if statement is true(): **..**
	elif statement is false(): **..**
	# else, Error
	Or
	if statement = true: **..**
	elif statement = false: **..**
	# else, Error

<is_available>{false}
	if NOT is_available;
	if is_available is not true();
	if is_available is bool() AND is_available is not true();
<is_available><>
	*if NOT is_available; generates **SE**;*
	*if is_available is not true(); **also gets implemented for <is_available><> ****
	if is_available is bool() AND is_available is not true(); **doesn’t get implemented, doesn’t show any error. ****

Syntax of if statement with bool()

<is_rainy>{true}
<has_umbrella>{false}

if is_rainy is true() AND has_umbrella is false();
if is_rainy AND NOT has_umbrella;

if is_rainy AND ~~not~~ has_umbrella;
*if is_rainy and ~~not~~ has_umbrella; both generate **SE**; (not-operator should be used only after is-operator)*
*if is_rainy and ~~NOT~~ has_umbrella; generates **SE**; (NOT-operator should be used only at the start of the statement)*

if is_rainy AND has_umbrella is false();
if is_rainy and has_umbrella is false(); means if both are false. So, it has wrong interpretation.

if is_rainy is true() AND NOT has_umbrella;
if has_umbrella is not true() AND is_rainy;

*if is_rainy is true() and ~~not~~ has_umbrella; generates **SE**; (not-operator should be used only after is-operator)*
*if has_umbrella is not true() and ~~is_rainy~~; generates **SE**; (here should be Comparison Function)*

Summary of Comparison Operators:

null()
    null.base()
    null.list()
bit.zero()
    num()
        int()
        float() - float(0) **float(0) works if var is float().**
    name()
        str() - of(char) **of(char) works if var is str().**
        [str.number(), str.alphabet(), str.upper(), str.lower()**]**
        char() **char() works independently.**
        [char.number(), char.alphabet(), char.upper(), char.lower()**]**
        snip()
    bool()
        true()
        false() **true() and false() work independently.**
    un()
        str.un()
        num.un()
        bool.un()
bit.one()
    list()
        nums()
            range()
        names()
    dict() **Simple dict, Defined dict, Layer has no Comparison Operator.**
        one()
        many()
        tree()
    set()

self() **works within updater type of function that also works as a generator.**
default() **works for keyword arguments.**

Variable <it>

- It gets generated if bool() returns true (It means Function within it was successful with two statements: return true as bool; return **..** as bool_it;)
- or if .bool. returns true as endResult. (It means previous Property was successful with two statements: return true as boolResult; return **..** as valueResult/posResult/keyResult/ endResult/listVar/dictVar/setVar;)

*It generates **SE** as simpleVar, mainVar.*
*If used outside of indentation as camouflagedVar or dependentVar, It generates **NameE**.*

<names.list><John, Yara, Sarah>
    if .get (1) .bool{names.list}: **Here, <it><John>**
        *<~~it~~><Jon>, .remove h<~~it~~>, ..remove h<~~it~~>{it} all generate **SE**;*
        <name>{it} del(it)
        .remove h<name>
    else:
        *<~~it~~><Jon> generates **SE**;*
        *<out>{it} generates **NameE**;*
    <out><first name is 'name'.> $ ~~first name is Jon.~~

    *if .get (1) .bool{names.list} AND .find h .bool{it}: **Here, <it><John> gets overwritten as <it>(3)***
        <name>{it} del(it)
        *.remove h<name> generates **VarE**;*

<name><John>
    if bool(remove_john_with_bool(name)): **Here, <it><> is null.base().**
        <out>{it} $ ~~(a line generated by /b(null).)~~

<nums.list>(3)
    if .get (1) .bool{nums.list} AND in_range(it, 1, 3):
        <out>{stringify(it**2)/db} ***Note: <out>(3**2/~~db~~) generates NameE;***
        <out>< is less than 10.>
    $ ~~9 is less than 10.~~

(e12)

if .get (2) .bool{couples.list} AND .get (1){it} = “John”:

Here, <it><John, Yara> is list().

<out><John\’s wife\’s name is /db>

..get (2)<out>{it}

$ ~~John’s wife’s name is Yara~~

(e8)

if .get apple .gather Earligold .bool{fruits.list} AND it is not null():

Here, <it><ripen: early> is one().

loop:

: for value in it AND key in keys

: <out><“‘key’”\: ‘value’>

$ ~~“ripen”: early~~

<str><Amazon>

if .find a .bool{str} AND .get (it-1) .bool{str}: **Here, <it>(3) gets overwritten as <it><m>**

<out>{it} $ ~~m~~

if .find A .bool{str} AND .get (it-1) .bool{str}: **Here, <it>(1) doesn’t get overwritten but whole condition doesn’t get implemented.**

<out>{it} **has no terminal interaction.**

*get names.list if .find John .bool{it} AND ~~it~~ < 5; generates **FSE**; (SE; names.list is p-list.)*

get names.list if .find John .bool{it} AND .find John{it} < 5; works.

def in_check():

get names.list;

if .find John .bool{names.list} AND it < 5:

return true;

else:

return false;

get names.list if in_check(it); also works.

String Syntax

Easier

- break and don't break
    - /b(space)
    - /b(null), /db(null) only with inOperator, outOperator.
    - By default, after printing command, outOperator breaks while inOperator doesn't.

<out><> **is** <out>
<out><Hi\,> **is** <out><Hi\,/b>
<name>{in}What is your name? {} **is** <name>{in}What is your name? /db{}
$
$ ~~Hi,~~
$ ~~What is your name?~~ John
$

<out><Hi\, /db>
<name>{in}What is your name?/b{}
or
<out><Hi\, What is your name?>
<name>{in}{}
$ ~~Hi, What is your name?~~
$ John
$

<out><Hi/db>
<out>< /db> **also as** <out>{space/db}
<out><John>
or
<temp><Hi>
<temp> += space
<temp> += <John>
<out>{temp}
$ ~~Hi John~~
$

(e12)

<out><we are '.get (2){couples.list}'.>
$ ~~we are John, Yara.~~
$
<out><we are /db>
..get (2)<out>{couples.list/db}
<out><.>
$ ~~we are <John, Yara>.~~
$

<tries>(3)
<out><To guess right number\,/b it took you 'tries' tries...>
$ ~~To guess right number,~~
$ ~~it took you 3 tries...~~
$

<detail><Hi\, My name is John./b I am 29 years old.>
<out>{detail}
$ ~~Hi, My name is John.~~
$ ~~I am 29 years old.~~
$

Variables

- null.base(), bool() generate error.
    - and, stringify() is not useful for null.base() while somewhat useful for bool().
      <none> would become string "(nullBase)".
      <true>, <statement>(3>1) would become string "(true)".

<statement><you have 'none' more tries.>
***and** <out><you are 'true'.> generate **DisE**;*
*<statement><Here|'s a statement|: {3>1)> generates **SE**;*

<statement><//Here’s a statement: (3>1)//>
<out>{statement}
$ ~~Here’s a statement: (3>1)~~

- un() generates error. But, stringify() is useful here.

<n>([])
<num>(2*n+3)

*<out><‘n’ is used in this complex number|: ‘num’> generates **DisE**;*
<out><‘stringify(n)’ is used in this complex number\: ‘stringify(num)’>
$ ~~n is used in this complex number: 3+2*n~~

*<num>{in}Type a number |’’a’|’ that makes this statement true {4>5)|: {} generates **NameE** than **SE**;*
<num>{in}Type a number \’a\’ that makes this statement true \(4\>5\)\: {}
$ ~~Type a number ‘a’ that makes this statement true (4>5):~~ 7

- name(), num() are allowed with string.

<name><Sarah> <greeting><Hi\, ‘name’>
<num>(13) <greeting><Hi\, ‘num’>

*<rating><rating is {-7.6).> generates **SE**;*
<rating><rating is -7.6.>

- locally, math generates error. But, stringify() is useful here.

<out><Cart total is|: {4999+120+99)>
<out><Cart total is|: ‘4999+120+99’>
*<out><Cart total is|: ‘item1_price+item2_price+99’> All three generate **SE**;*
<out><Cart total is\: ‘stringify(4999+120+99)’>
$ ~~Cart total is: 5218~~

- 1d-nums(), 1d-names() are allowed with string.

<nums.list>(5, 2, 1, 5, 7, 1)
<out><Max from ‘nums.list’ is ‘.lambda,filter a, max{nums.list}’.>
$ ~~Max from 5, 2, 1, 5, 7, 1 is 7.~~

- locally, 1d-nums(), 1d-names() generate error. But, stringify() is useful here.

*<statement><our names are|: ≺John, Yara>> generates **SE;***
<statement><our names are\: ‘stringify(&″John″&″Yara″)’>
<out>{statement}
$ ~~our names are: John, Yara~~

- Rest of the bit.one() generates error and stringify() is not useful here.
    - It returns “(nullList)”, “(2dList)”, “(oneDict)”, “(manyDict)”, “(set)”.

Syntax of selfArgument

- **Use variable, function inside round bracket.**
  - Variable used here is called dependentVar which also supports valueDependent properties.
  - Generator type of function is supported.
    Note: If function has multiple returns, it tries to generate a single variable.

- **Locally base(), num.un(), bool.un(), nums()(>1), range() are supported.**
  - Use str() floated and put a backslash before special character.
  - Use others inside round bracket.

<name><John>
<detail.list><>
<chars.list><>
<nums.list><>
<availability.list><>

.add name, (name)<detail.list>
.add age, (date("y", 29))<detail.list>
§ ~~<name: John age: 29>~~

.append (halfstr(name))<chars.list>
.append,nis (halfstr("Yara"))<chars.list>
§ ~~<<Jo, hn>, Ya, ra>~~

.append (0)<nums.list> **here, selfArgument is num(), not nums()(=1).**
.append (1:1)<nums.list> **here, local range() is used as nums()(=1).**
.append,nis (2, 3)<nums.list>
.append (.len{name})<nums.list>
.append (double(2)+1)<nums.list>
.append ((double(2)-1)*2)<nums.list>
<null.list><> .append (null.list)<nums.list>
*here, .append ()<nums.list> generates **SE;***
§ ~~<(0, (1), 2, 3, 4, 5, 6, ())>~~

.pop,nis (2, pos(nums.list, -1))<nums.list>
*here, .pop,nis (2)<nums.list> generates **ASE**;*
§ ~~<(0, 2, 3, 4, 5, 6)>~~

.lambda,filter a, (a%2=0)<nums.list> **filters num() divisible by (2).**
§ ~~<(2, 4, 6)>~~

(e11)

if .find John .bool{names.list}:
    .append **(3>2)**<availability.list>
else:
    .append (false)<availability.list>
*here, .append (.find John .bool{names.list})<availability.list> generates **PSE**; (.bool. used as valueDependent.)*
<out>{availability.list}
$ ~~<(true)>~~

<detail><My name is John.>
<name><John>
<detail.list><>

if .find \. .bool{detail} **AND** it != .len{detail}:
    <out><More than one sentences...>
No terminal interaction.

- **Local str() doesn't allow any combinations. (except with <space>)**

if .find \.(space) .bool{detail}:
    <out><More than one sentences...>
No terminal interaction.

.find (name)┼.<out>{detail}
*.append My name is {name)<detail.list> Both generate **SE**;*

- **Locally, null.list(), names(), 2d nums(), dict(), set() aren't allowed.**

<names.list><>
<reversed_nums.list><>

<temp.list><John, Yara>
.append,nis (temp.list)<names.list> § ~~<John, Yara>~~

*here, .append,nis (John, Yara)<names.list> generates **NameE**; (var <John> doesn't exist)*
*.append,nis John, Yara<names.list> generates **ASE**;*

<temp.list>((1:3, (4:6))
.insert,nis (1), (temp.list)<reversed_nums.list> § ~~<((4, 5, 6), (1, 2, 3))>~~

*here, .append,nis ((4:6), (1:3))<reversed_nums.list> generates **SE**;*

Main exhaust using * operator

- **selfArgument, Member of list(), Value of key and Input of function support it.**

- **Following properties may benefit from it:**

    - simpleDot:
        - 2nd selfArgument of .replace., .replace,pos.
        - 2nd selfArgument of .add. on dict(), .change.
        - Both selfArguments/ Only selfArgument of .add,nis.
        - 1st selfArgument of .add,keys. on null.list().
          2nd selfArgument of .add,keys. **(But still have to copy keysToBeAdded-1 times)**
        - Both selfArguments of .add. on set(), .add,set.

    - simpleDot & nextDot:
        - selfArgument of .append., .append,nis.
        - 2nd selfArgument of .insert., .insert,nis.

<detail><My name is John. I am 29 years old.>
<details.list><>
.append (*detail)<details.list>
***Note**: .append<details.list>{*detail} generates **SE**; Camouflage has **no need** to support this syntax.*

*Now, <out>{detail} generates **NameE**;*

*.append *My name is John<details.list> generates **SE**;*
.append *My name is John<details.list> is not main exhaust.

<num>(2147483648)
<nums.list><'*num', (1)> **or** <nums.list>(*num, 1)
or <nums.list>(1) .insert (1), (*num)<nums.list>
or <nums.list><> .append,nis (*num, 1)<nums.list>

*.append,nis ⁺(1, 3, 5, 7, 9)<details.list> generates **SE**;*
*.append (1*num)<nums.list> is not main exhaust.*

(e11, e12, e8)

<names.list><'names.list'> **or** <names.list><'*names.list'>

or <temp.list><>
.append (*names.list)<temp.list>
<names.list>{temp.list} del(temp.list)
§ ~~<<Sarah, Mosh, John, Mia, Yara, Sarah, Bob>>~~

<couples.list><couples: 'couples.list'> **or** <couples.list><couples: '*couples.list'>

or <temp.list><>
.add couples, (*couples.list)<temp.list>
<couples.list>{temp.list} del(temp.list)
§ ~~<couples: <<Sarah, Mosh>, <John, Yara>, <Mia, Bob>>>~~

***Note**: .add couples, (couples.list)<couples.list> generates **VPME**;*
*.add couples, (⁺*couples.list)<~~couples.list~~> generates **DisE**; (main exhaust can't dissolve <couples.list>)*
*.add (.get (1){couples.list}), (.get (2){couples.list})<couples.list> generates **VPME**;*
*.add (|⁺*couples.list)<~~couples.list~~> generates **DisE**; (main exhaust can't dissolve <couples.list>)*

.get apple<fruits.list>
Here, .get. gives value-Layer(>1)

<apples.list><apple: '*fruits.list'>
or
<apples.list><>
.add apple, (*fruits.list)<apples.list>

<fruits.list><apple: <Cameo : Jonagold> mangoes: <Kesar : Gir Kesar>>

<apples.list><Cameo : Jonagold>

<mangoes.list><Kesar : Gir Kesar>

<fruits.list><apple: ‘*apples.list’ mangoes: ‘*mangoes.list’> **is more readable.**

(e11)

def fun():

get names.list++; **It doesn’t make a copy if input uses main exhaust.**

get none;

..find,all Sarah<all.list>{names.list}

.remove,pos (.get (-1){all.list})<names.list>

.exchange (.find Mia{names.list}), (.find Yara{names.list})<names.list>

return names.list;

fun(*names.list : names.list)

§ ~~<Sarah, Mosh, John, Yara, Mia, Bob>~~

Sharing using | operator

Camouflage can share values across members. So does Value of key. So, they don’t support this syntax:

(e12)

<lst.list>{couples.list} **It has length (3).**

.insert (1), (0)<lst.list>

or <lst.list><(0), ‘|couples.list’> § ~~<(0), <Sarah, Mosh>, <John, Yara>, <Mia, Bob>>~~

<dct.list><couples: ‘couples.list’> **Value of key “couples” has length of (3).**

or <dct.list><couples: <|‘couples.list’>> § ~~<couples: <<Sarah, Mosh>, <John, Yara>, <Mia, Bob>>>~~

*Here, <lst.list>{¦couples.list} **and** <dct.list><couples: ‘¦couples.list’> generate **SE**;*

It works if Variable is going to be p-list:

<names.list><John, Yara>

<out><'names.list'> $ ~~John, Yara~~
*<out><'|names.list'> generates **SE**; (String Syntax doesn't allow Sharing.)*
<any(lst)><'names.list'> § ~~John, Yara~~
*<any(lst)><'|names.list'> generates **SE**;* **(or works as ~~<John, Yara>~~)**

<any(lst)><'names.list', (0)> § ~~<<John, Yara>, (0)>~~
<any(lst)><'|names.list', (0)> § ~~<John, Yara, (0)>~~

It shares only nums() or names():

<name.list><John>
<couple.list><John, Yara>

<name.list><'name.list'> **or** <name.list><'.pop{couple.list}'> § ~~<<John>>~~
<name.list><'|name.list'> **or** <name.list><'|.pop{couple.list}'> § ~~<John>~~

<name.list><'.pop{name.list}'> § ~~<<>>~~
*<name.list><'~~|.pop{name.list}~~'> generates **DisE**;*

<nums.list>(1, 2, 3)
<dct.list><nums: <((|*nums.list), 4)>> § ~~<nums: <(1, 2, 3, 4)>>~~
vs <dct.list><nums: <'|*nums.list', 'true'>> § ~~<nums: <(1), (2), (3), (true)>>~~

<name><John>

*<dct.list><name: <'~~|name~~'>> generates **DisE**;*
<dct.list><name: |'name'> § ~~<name: |John>~~
and** <dct.list><name: <|'name'>> § ~~<name: <|John>>~~ **are part of String Syntax.

<dct.list><name: 'ofStr(name, 1)'> (of **easyA** env.)
or <dct.list><name: <'|ofStr(name, 1)'>>
§ ~~<name: <J, o, h, n>>~~

<children.list><name: John wife: Yara,
name: Mia>

<dct.list><name: Sarah, '~~children.list~~'>
and *<dct.list><name: Sarah, '~~\|children.list~~'> generate* ***DisE****;*

<dct.list><key: 'children.list'>
§ ~~<key: <name: John wife: Yara,~~
~~name: Mia>>~~

<dct.list><key: <'~~children.list~~'>
and *<dct.list><key: <'~~\|children.list~~'>> generate* ***DisE****;*

How to use it with selfArgument:

<name.list><John>

.append (name.list)<name.list> § ~~<John, <John>>~~
.append (.get (1){name.list})<name.list> **or** .append (|name.list)<name.list> § ~~<John, John>~~

(e11)
<update.list><John, Jon>

.replace (.get (1){update.list}), (.get (2){update.list})<names.list>
or .replace (|update.list)<names.list> **Here also, .replace. gets two selfArguments.**

<sarah.list><Sarah, <John, Mia>>
<john.list><John, Sarah>
<children.list><>

.add (|*sarah.list)<children.list>
.add (|*john.list)<children.list>
§ ~~<Sarah: <John, Mia> John: Sarah>~~

(e5)

```
        <path.list><'.find John{family_tree.list}', linked, daughter>
        ..get (|path.list)<out>{family_tree.list}
        del(path.list)
        $ ~~Sarah~~

<dct.list><Sarah: (12)  Mosh: (13)>

        <name>{in} Enter your name followed by your lucky number\: {}  $ John 14.
        if .count (space){name} = 1:
                if .get (.get (1:.find (space){name}-1){name}) .bool{dct.list}:
                        # B gives you (5). U gives you "John".
                        .change (|ofStr(name, space, mixed= true))<dct.list>
                else:
                        .add (|ofStr(name, space, mixed= true))<dct.list> is .add John,
                        (14.0)<dct.list>
```

How to use it with functions:

```
<nums.list>(1, 2, 3, 5, 8, 13)

        def sum():
                loop:
                        ; <sum>(0)
                        : get num;
                        <sum> += num
                until:
                        if get none:
                                get none;
                                return sum;

        <out>{sum(|nums.list)}  $ ~~(32)~~ gets 6 arguments.
        <out>{sum(|nums.list, 21)}  $ ~~(53)~~ gets 6+1=7 arguments.
        <out>{sum(|join(nums.list, 21))}  $ ~~(53)~~ gets 7 arguments.
```

<name><John>

```
def simple():
    get str if it is str();
    get none;
    return str;

simple(name : pack(names.list))  § <John>
<out>{simple(name)}  $ John


def split():
    get str if it is str();
    get none;
    return |ofStr(str, 1);
    is loop:
        : for char in str
        : return char;

split(name : pack(chars.list))  § <J, o, h, n>
<out>{split(name)}  $ <J, o, h, n>
```

*Here, return |str; statement generates **FSE**; (SE)*

Syntax interpretation as we type

<var>{

- **It allows bit.zero() except str.un().**
  <n>([]) § ~~(n)~~ is str.un()
  *<max>{n} generates **VarE**;* **Exception**: <out>{n} $ ~~(n)~~ works
  <max>(n) § ~~(n)~~ is num.un()
  <min>{max} § ~~(n)~~ is num.un()

<var>{none} is null.base()
<var>{space} is str()
<var>{date("y", 30)} is Snip
<var>{true} is bool()
<var>{num} has <num>(3) is num()

*<var>{var.list} has <var.list><> generates **VarE**;*
*<var>{names} has <any(names)><John, Yara> generates **VarE**;*

<var.list>{

- **It allows bit.zero() except un() (null.base() becomes null.list() while with base(), it is packing.)**
  *<var.list>{max} generates **VarE**;*
- **null.list(), list()**
- **dict() (including tree().)**
- **set()**

<var.list>{none} § ~~<>~~ is null.list()
<var.list>{true} § ~~<(true)>~~ is names()
(e5) <var.list>{family_tree.list} is tree()

<any(var)>{

- **It becomes p-none if null.base(), base(), un() except str.un()**
  <any(var)>{max} § ~~(n)~~ is num.un()
- **p-list if null.list(), list(), dict() (including tree()), set()**

<any(var)>{none} is null.base()
<any(var)>{null.list} has <null.list>{none} is null.list()
<any(var)>{true} is bool()
(e31) <any(set)>{set.list} is set()

<num>(

- **It becomes num(), un(), bool()**

- **It can dissolve num(), str.un(), num.un()**
  *<var>(none), <var>(true), <var>(space) generate **DisE**;*

*<num>() generates **SE**;*

<n>([]) is str.un()

<num>((**becomes complex-math. It can dissolve num(), str.un(), num.un()**

<num>(- **works**

<num>(-3*3) ~ <num>(-9) is num()
<num>(-num*num), <num>((-num)*num) has <num>(3) ~ <num>(-9) is num()
<num>(-n*n), <num>((-n)*n) has <n>([]) ~ <num>(-n**2) is num.un()

<statement>(-6<0) is bool() as true
<statement>(-num*2<n), <statement>((-num)*2<n) has <n>([]), <num>(n-7) ~ <statement>(14-(2*n)<n) is bool.un()

*<num>(~~.~~7) generates **SE**;*

<num>(.len{nums.list}) has <nums.list>(1, 2, 3) ~ <num>(3) is num()

(e8) <num>(.get apple .len{fruits.list}), <num>(.get apple; .len{fruits.list}) ~ <num>(8) is num()

<num>(3) is num()
<num>(num+1) has <num>(3) is num()

<statement>(num>n) has <n>([]), <num>(n) is bool() as false
<statement>(n!=5) has <n>([]) is bool.un()

<str><

- **It becomes null.base(), str()**

<str><> is null.base()
  - <out><> ~ <out> has string syntax

*<str><≤ generates **SE**;*
*<str><{3)> generates **SE**;*

<str><\(3\)> <out>{str}
$ ~~(3)~~ is str() with len (3)
<str></b Hi\, Welcome.> <out>{str}
$ **(A line generated by /b(space))**
$ ~~Hi, Welcome.~~ is str() with len (12)

<str><' **can dissolve according to String Syntax**

*<str><'true'> generates **DisE**;*
<str><'names.list'> has <names.list><John, Yara> <out>{str}
$ ~~John, Yara~~ is str() with len (10)

<str><J> is str() with len (1)

<nums.list>(

- **It becomes nums()**
  It can dissolve num()
  It can dissolve 1d-nums(), null.list() as well with use of another set of round bracket

- **If whole, it becomes range()**
  Here, it can dissolve num()
  <range.list>(1:.len{str}) has <str><Mia> ~ <range.list>(1:3) is range()
  *<range.list>(1:3; 4) generates **SE**;*

<nums.list>(double(num), double(num)-2, (.reverse{range.list})) has <num>(3) ~ <nums.list>(6, 4, (3, 2, 1)) is 2d-nums() with third member as nums()

*<nums.list>() generates **SE**;*

<nums.list>((

- **It becomes 2d-nums()**

- **Here, first member becomes null.list()**
- **1d-nums() (It can dissolve num(). If whole, It can dissolve 1d-nums(), null.list() as well)**
- **If whole, range() (Here, it can dissolve num())**

  <nums.list>(()) ~ <nums.list><<>>
  <nums.list>((null.list)) has <null.list><> ~ <nums.list><<>> are **2d-names() (is exception!)** with first (and only) member as null.list()

  <nums.list>((num, -num)) has <num>(1) ~ <nums.list>((1, -1))
  <nums.list>((one_to_three())) ~ <nums.list>((1, 2, 3)) are **2d-nums()** with first (and only) member as nums()

  *<nums.list>((one_to_three(), 4)) generates **DisE**; (needs whole set of round bracket.)*
  *<nums.list>(one_to_three()) generates **DisE**; (needs another set of round bracket.)*

<nums.list>((1:3)) here, member is range()
*<nums.list>((1:3, 4)) generates **SE**; (set of round bracket should be whole.)*

*<nums.list>(((1:3), 4)) generates **SE**; (here, **third bracket must be for complex-math**.)*

<nums.list>((1:3), 4) has length (2)

- **If second bracket is for complex-math, it remains 1d-nums()**
  As complex-math, it can dissolve num()
  <nums.list>((num+4)*2) has <num>(2) ~ <nums.list>(6*2) ~ <nums.list>(12)

<nums.list>(((**Here, third bracket is for complex-math. It can dissolve num()**

<nums.list>(((2+4)*num, 13)) has <num>(2) ~ <nums.list>((6*2, 13) ~ <nums.list>((12, 13) is 2d-nums()

<nums.list>(((2+4)*2)+3, 16) ~ <nums.list>((6*2)+3, 16) ~ <nums.list>(12+3, 16) ~ <nums.list>(15, 16) **is 1d-nums() because second bracket is also for complex-math**

<names.list><

- **It can become null.list(), nums(), names(), dict(), set()**

<null.list><> is null.list()

<names.list><(
<names.list><((

- **Here, first round bracket is for member num().**
  Second one is for complex-math.

- **Both can dissolve only num()**

*<null.list><(}> generates **SE**;* (While <names.list><\(\)> is names())
*<range.list><(1:3)>, <nums.list><(1; 2, 3)> generates **SE**;*

<nums.list><((3-num)*1), (num), (num+1), (double(2))> has <num>(2) ~ <nums.list>(1, 2, 3, 4) is nums() (Here, member 'double(2)' also works. *Member double(2) generates **SE**;*)

*<names.list><(num-1), num> ~ <names.list><(1), **num**> is names()*

<names.list><<

- **Here, first member becomes null.list(), 1d-nums(), 1d-names()**

<names.list><<>> is names()
<nums.list><(1), (2), <>> ~ <nums.list>(1, 2, ()) is nums(). In both examples null.list() as a member is present at first and third position respectively.

<names.list><<(

*<names.list><<(}>> generates **SE**;*

- **Here, round bracket is for member (num()) of first member.**
  It is ideal for 1d-names() as <<(num1), (num2), str>> or <<(num1), str, (num2)>>
- **It can be for whole first member (nums(), range()) as well**
  It is ideal for 1d-nums() as <<(num1, num2)>, secondMember> and <<(num1:num2)>, secondMember>
- **Note: It can't be for complex-math**

- **In both ways, It can dissolve num() only**

  <names.list><<(num), John>, Mia> has <num>(0) ~ <names.list><<(0), John>, Mia>
  Here, first member is names() with len (2)

  <nums.list><<(num), (-num)>>, <nums.list><<(num, -num)>> has <num>(1) ~ <nums.list>((1, -1)) is 2d-nums() with first member with len (2)

  *<nums.list><<(1, -1); John>>, <nums.list><<(1:-1); John>> Both generate **SE**;*

<names.list><<((**Here, second round bracket is for complex-math. It can dissolve num()**

<nums.list><<((num-2)*1)>> has <num>(3) ~ <nums.list>((1)) here, 2d-nums() has first member with len (1)

*<nums.list><<(3-2)*1>> generates **SE**;*
*<nums.list><<({3})>>, <nums.list><<({3-2})>> In both examples, "(" generates **SE** due to unexpected ")";*

*<nums.list><<((temp.list))>> generates **DisE**;*

<nums.list><<((num-2)*1, -1)>> ~ <nums.list>((1, -1))
<nums.list><<((num-2)*1:-1)>> ~ <nums.list>((1, 0, -1))

- **Bracket of member nums() if used as whole, can dissolve 1d-nums(), null.list() as well**
  It is used as <<(existing_var)>, secondMember>

  <nums.list><(1), (2), <(num, num+1)>> has <num>(3) ~ <nums.list>(1, 2, (3, 4))

  <nums.list><(1), (2), <(temp.list)>> has <temp.list>(3, 4) ~ <nums.list>(1, 2, (3, 4))
  <nums.list><(1), (2), <(null.list)>> has <null.list><> ~ <nums.list>(1, 2, ())

  *<nums.list><(1), (2), <(temp.list), John>> generates **DisE**; (bracket for num() can dissolve num() only.)*
  *<nums.list><(1), (2), <(temp.list, 5)>> generates **DisE**; (bracket of member nums() should be whole to dissolve nums(), null.list().)*

<names.list><<'

- **It can dissolve base() only**

<nums.list><(0), <'num'>> has <num>(1) ~ **<nums.list>(0, (1))** is nums()
<names.list><(0), <'true'>> § **<(0), <(true)>>** is names()

<names.list><(0), <John>> is names()

<names.list><‘

- **It can become nums(), names(), dict(), set()**
- **It can dissolve base(), null.list(), 1d-nums(), 1d-names(), key or value of dict(), value of set()**

<names.list><‘null.list’> has <null.list><> ~ <names.list><<>> is names()
<names.list><‘null.list’, ‘lst.list’, ‘couple’> also has <lst.list>(1, 2), <any(couple)><John, Yara> ~ <names.list><<>, <(1, 2)>, <John, Yara>>

<nums.list><(0), ‘lst.list’> ~ <nums.list>(0, (1, 2))
*<nums.list><(0), <‘lst.list’>> generates **DisE;***

<names.list><‘true’, ‘space’> § <(true), **(space)**> is names()

<dct.list><‘key’: ‘value’> has <key><name>, <value><John> ~ <dct.list><name: John> is dict()
If it has <key><name is>, <value><John> ~ <dct.list><**name%is**: John>

<set.list><‘date(“y”, 30)’ : 30> is set()

<names.list><John> is names()
<names.list><name: John> is dict()
<names.list><John : Jon> is set()

Within Single quote, use of Local num(), Augmented Assignment Operators or Comparison Operators generates SE:

*<str><'3+1'>, <availabilities.list><'3>1'> generates **SE**;*
*<any(var)><'num+1'>, <any(var)><'num>1'> has <num>(3) generates **SE**;*

p-list/ p-any doesn't allow use of Comparison Operators within Round bracket:

*<var.list>(3>1), <any(var)>(3>1), <dct.list><available: (3>1)> generate **SE**;*
Note: <var>(3>1) is bool() as true

*<var.list>(1, 2, <>), <any(var)>(1, 2, <>) generate **SE**;*
***Note:** <var>(1, 2, <>) generates **SE**;*

p-list/ p-any generates either SE or DisE with Type un():

<n>([]) is str.un()
*<n.list>([]), <any(n)>([]) generate **SE**;*

<num>(3+n) is num.un()
*<var.list>(3+n), <any(var)>(n+3) generate **DisE**; (p-list/ p-any can't dissolve un().)*

<statement>(num>2) is bool.un()
*<var.list>(num>2), <any(var)>(num>2) generate **DisE**; (p-list/ p-any can't dissolve un().)*

names(), dict() vs set():

<var.list><key> is names()
<var.list><key, value> is names()
<var.list><key: value> is dict()
<var.list><key : value> is set()

<var.list><key is> is names()
<var.list><key is, value> is names()
<var.list><key is : value> is set()

*<var.list><key-is: value> generates **SE**; (not SE as :)*
<var.list><key\ is: value>, <var.list><key'space'is: value> works.
<var><key is> <var.list><'var': value>, <var.list><name: John 'var': value> works.

<var.list><name: John age is: (29)>
..get name<out>{var.list} $ ~~John age~~
..get is<out>{var.list} $ ~~(29)~~

*<var.list><names: <John, Yara> age-difference: (2)> generates **SE**;*

<var.list><'true'>, <var.list><<'true'>>, <var.list><'true', 'false'> are names()
*<var.list><'true': value>, <var.list><'true' : value> generate **DisE**; (not SE as :)*

DisE while dissolving dict(), set() as a member:

<john.list><name: John wife: Yara>

<family_detail.list><name: Sarah,
'john.list',
*name: Mia> generates **DisE**; (member of dict() can dissolve **key, value of key only**.)*

<family_detail.list><name: Sarah,
name: '.get name{john.list}' wife: '.get wife{john.list}',
name: Mia> works.

<family_detail.list><name: Sarah, name: Mia>
.insert (2), (john.list)<family_detail.list> also works.

<new_entry.list><(3) : Three, Five : (5)>

<set.list><(1) : One,
*'new_entry.list'> generates **DisE**; (member of set() can dissolve **value of set only**.)*

<set.list><(1) : One,
(.get Three{new_entry.list}) : Three,
'.get (5){new_entry.list}' : (5)> works.

Here, member '.get Three{new_entry.list}' works.
*But member (.get (5){new_entry.list}) generates **DisE**;*
<set.list><(1) : One>
.add,nis (new_entry.list)<set.list> also works.

Defined dict():

<layer.list><ckey: <~>>
*<defined.list><name: <~>> Both generate **SE**;*
// pack_ckey("name")

*<layer.list><ckey: 'null.list'> has <null.list><> generates **DisE**;*

*<defined.list><~~n~~ame: John age: (29)> generates **SE**;*
// pack_ckey("age")
*Here, pack_ckey("names") generates **FIE**; (key "names" doesn't exist.)*
*pack_ckey(3) generates **FIE**; (invalid argument.)*

<defined.list><name: John age: (29)>
update_ckey(defined.list, "age") works.
*Here, update_ckey(defined.list, "names"), update_ckey(defined.list, 3) generate **FIE**;*

*<dct.list><(1): John (2): Yara> generates **SE**; (not FIE, not SE as ~~(1)~~)*
// ~~pack_ckey~~(1)

Extras

How to address pointers:

There are three pointers. We can address them as p-none, p-list or p-any. For example, let's make a p-any that stores names.
<any(names)><John, Yara>

How to create/ modify Variable:

- Any of the following lines if Variable doesn't exist, creates it and assigns value to it. Else, changes its value.

  <var><value>
  <var>{existing_var} **also <var>{in}{}**
  ..property<var>{existing_var}
  <var>{generator()} **or** generator(none : var)
  <var>{generator(existing_var)} **or** generator(existing_var : var)

- *But any of the following lines **works only if Variable exists**. Else, generates **NameE**;*

  .property<var>
  .property<var>{existing_var}
  <var> += 1
  <var> += space
  updater(var)

Use of _:

<any(exam_detail)><starts_at: 'time("hh", 9)' total_seconds: ((3_600*3)-30*60)> **uses _ to declare a Variable, to define a key, to ease reading numbers while reviewing your code.**
<num>(10_000)
fun(input=1_00_000)

Use of //:

<greeting><//Hi, 'name'.//> **is** <greeting><Hi\, \'name\'.>

(e12) ..find (x)<out>{couples.list} $ ~~(2)~~
// <any(x)><John, Yara>

(e11) ..find<out>{names.list} $ ~~(2)~~
// <>{.find.}
(startsWith) "M"
<>{}

<dict.list><name: Sarah>
// pack_ckey("name")

<bool>{in}Do you agree? {}
// t.len(1, 1)
if bool = "y" or "Y":
<out>{agreed!}
elif bool = "n" or "N":
<out>{not agreed!}
else:
<bool>{in}Do you agree? y\/n {}
// t.len(1, 1)

Use of &:

none $ **(a line generated by /b(null))**
&none $ ~~<>~~
&&none $ ~~<<>>~~
and &&none*3 **[or as** &&*3 none**]** $ ~~<<>, <>, <>>~~ **are fnull().**

& "Sarah" & "Mosh" $ ~~<Sarah, Mosh>~~ **is fstr().**
&! "Sarah" & true $ ~~<Sarah, (true)>~~ **is just names().**

<couple1.list><John, Yara>
<couple2.list><Mia, Bob>
&& couple1.list && couple2.list $ ~~<<John, Yara>, <Mia, Bob>>~~ **is fnames().**
& "Sarah" & "Mosh" &| couple1.list &| couple2.list $ ~~<Sarah, Mosh, John, Yara, Mia, Bob>~~ **is fstr().**

<john.list><John, (29)>
<mia.list><Mia, (25)>
<children.list><John, Mia>
<ages.list>(29, 25)
&& john.list && mia.list $ ~~<<John, (29)>, <Mia, (25)>>~~ **is also fnames().**
&!| john.list &| mia.list $ ~~<John, (29), Mia, (25)>~~ **is just names().**
&&! children.list && ages.list $ ~~<<John, Mia>, <(29, 25)>>~~ **is just 2d-names().**

<nums.list>(3:5)
&1 &2 &| nums.list $ ~~<(1, 2, 3, 4, 5)>~~ **is fint().**

&!1.0 &2 &| nums.list $ ~~<(1.0, 2, 3, 4, 5)>~~
and &!1 &2 & nums.list $ ~~<(1, 2, (3, 4, 5))>~~
and &!0 &true $ ~~<(0), (true)>~~ **are just nums().**

Use of #:

this line is a comment.
<out><Hello\, World!>
#! this line and everything below it are comments...

null() (len =0)	null.base()		
	null.list()		
bit.zero()	null.base()		
	base()	num()	int()
			float()
		name()	str()
			Snip
		bool()	
	un()	str.un()	
		num.un()	
		bool.un()	
bit.one()	list()	null.list()	
		nums()	
		names()	
	dict()	one() (len =1)	Simple-dict/ Defined-dict (including Layer)
		many() (len >1)	Simple-dict/ Defined-dict (including Layer, tree())
	set()		