Properties

.random. and .random,all.
.shift. .exchange. .sort. and .reverse.

.pop. and .pop,nis. with list()
.pop (pos). with dict()
.pop (key). with dict(=1)

.append. .append,nis. .insert. and .insert,nis.
How to use them with Boolean bundle .get (pos) .bool{var} and some examples
.append. .insert. with dict()
.append. .insert. with Layer

pack(), join(), unpack() and similar actions by properties
join(), unpack() with dict()

.get,pair.

.lambda,new.
.lambda,reduce.
.lambda,filter.
.lambda,spread.

(e1), (e2), (e3) are dict(=1)
(e4) is dict(>1)
(e5), (e6), (e7) are tree()
(e8) is Layer
(e9), (e10) are dict(=1)

(e21), (e23) are str()
(e22) is num()

(e11), (e12) are names()
(e31) is set()

Introduction

Variables that support properties:

nullVar (nullBase, nullList)
bitZero (nullBase, strVar)
listVar (nullList, namesVar, numsVar)
dictVar (oneDict, manyDict)
setVar

Properties can be:

simpleDot **.**
previousDot **C..**
nextDot **C.**
dependentDot **D** (valueDependent **Dv**, rfrDependent **Dr**)

> **append:zero and insert:zero work as simpleDot and nextDot. As nextDot, no other properties are supported.**
> **add:zero, change, rename and range work only as simpleDot.**
> **get:zero, replace:zero, remove:zero, shift, exchange, sort, reverse, pop, random,all and lambda:zero also work as simpleDot.**
>
> **bool works only as rfrDependent.**

Types of Arguments:

- **selfArgument** can't be null.base(), str.un(), tree(). **They generate VarE; (Valueholder <none> generates SE;)**
  len (with set()), bool, sort, reverse, random:zero, append:zero (as nextDot) accept no selfArgument.
  pop (with list(), dict(>1)) doesn't generate NAE if there is no selfArgument.

Note: Property's get none as selfArgument; statement means it receives no input.

2d-list(), dict(), set() are needed as selfArgument:

<couple.list><couple: <<Sarah, Mosh>>>
<temp.list><<John, Yara>, <Mia, Bob>>
.get couple .append,nis (temp.list)<couple.list>
§ ~~<couple: <<Sarah, Mosh>, <John, Yara>, <Mia, Bob>>>~~

<fruits.list><apple: <Cameo : Jonagold>>
<temp.list><Kesar : Gir Kesar>
.add mangoes, (temp.list)<fruits.list>
§ ~~<apple: <Cameo : Jonagold>, mangoes: <Kesar : Gir Kesar>>~~

- **valueArgument** is base().
  It is given by get.
  It is accepted by find:zero, count, replace,all, remove,all, fetch:zero with dict(=1).
- **posArgument** is int(>0).
  It is given by find.
  It is accepted by get (with Variables other than str()), replace,pos, remove,pos, replace,num, replace,lst, shift, exchange, pop.
- **keyArgument** is str(), num().
  It is given by fetch. (with dict(=1)) (Note that it never gives “linked” key.)
  It is accepted by get, get,pair, pop.

Note: len, gather, sort, reverse, add:zero, change, rename, range, random:zero and lambda:zero accepts no valueArgument, posArgument or keyArgument.
Property bool accepts all three. It also accepts boolArgument and endArgument as well.

- **boolArgument** is bool().
  It is given by
  - get (with valueResult, list(), dict(), set()),
  - gather (with dict(=1), null.list()),
  - find (with posResult),
  - fetch (with valueResult) (with endResult, if used with one selfArgument on set()).

  It is only accepted by bool.

Note: Since find and fetch generate PSE if used as simpleDot, **simpleDot Properties can’t get posArgument or keyArgument.**

Properties that give boolResult, don’t give it as simpleDot. **So, simpleDot Properties can’t get boolArgument as well.**

- **endArgument** is of any type except un(), tree().
  It is only accepted by Property bool and Variable.

  endArgument can be bool(), null.base():
  <name><John>
  if .find John .bool{name}: **Here, Property bool returns <true> as endResult.**
  .remove John<name> **is similar to <name>{none} because Property remove gives null.base() to Variable.**

- **previousArgument** is of any type except un(), tree().
  previousVar is of any type.

  <names.list><> **or** <names.list>{none} § ~~<>~~
  ..remove John<names.list>{name} **is similar to <names.list>{none}** § ~~<>~~

  .append<names.list>{none} generates **ADE**;
  *..remove John .append<names.list>{name} generates **ADE**;*

Properties as rfrDependent:

(e11), (e9), (e8), (e5)

if .len{names.list} <= 7;
<out><My name is ‘.get (3){names.list}’.> $ ~~My name is John.~~
<tmp><My name is ‘.get (3); .remove h{names.list}’.>
<out>{tmp} $ ~~My name is Jon.~~

<answer>{in} Type any movie released on same year as ‘.get (2001), (1)
{movies_release.list}’\: {} $ ~~Type any movie released on same year as fellowship of ring:~~
ocean’s eleven

```
if .gather Cameo .get taste{fruits.list} = “sweet”;
if .find Cameo .get taste{fruits.list} = “sweet”;
```

```
<out><John\’s ‘.get (3), linked .fetch Sarah{family_tree.list}’ is Sarah.>
```

$ ~~John’s mother is Sarah.~~

```
..get (3), linked .fetch,all Sarah<tmp.list>{family_tree.list}
<out><John\’s ‘.get (-1){tmp.list}’ is also Sarah.>
```

$ ~~John’s daughter is also Sarah.~~

<out><‘.random{names.list}’\’s turn…> $ ~~John’s turn…~~

- **get, gather, find, fetch, random can work as rfrDependent.**

get,pair, find,all, replace:zero and remove:zero, fetch,all, random,all generate PSE as rfrDependent.

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

```
if sum(nums.list) > 15;
```

or ..lambda,reduce a, (a+a)<num>{nums.list}

```
if num > 15;
```

- ***lambda:zero: generates PSE as rfrDependent.***

```
..sort; ..lambda,new x, (x+y); ..pop; ..reverse<new_nums.list>{nums.list}
```

Action of first ; is endResult to nums() ~~*<(1, 3, 5, 7)>*~~

Action of second ; is endResult to nums() ~~*<(4, 8, 12, 7)>*~~

Action of third ; is endResult to nums() ~~*<(4, 8, 12)>*~~

.reverse. gives endResult ~~*<(12, 8, 4)>*~~ *to Variable*

- ***sort, reverse, pop:zero: generate PSE as rfrDependent.***

Note: shift, exchange generate PSE as both rfrDependent and valueDependent.

Properties as nextDot:

- You can use any one of the following:
  - .append.
  - .insert.
  - .append,nis.
  - .insert,nis.

- You can’t use ; operator with nextDot properties.

(e12)
<lst.list><Sarah, Mosh>

```
..get (2)<lst.list>{couples.list} rewrites on existing <lst.list>.
<out>{lst.list}  $ ~~<John, Yara>~~

..get (2) .append,nis<lst.list>{couples.list}
<out>{lst.list}  $ ~~<Sarah, Mosh, John, Yara>~~
```

Which properties can modify a Variable:

For <any(var)><..>,

```
.property<var>
.property<var>{another_var}
```

- **simpleDot and nextDot properties can modify <var>.**

```
..property<any(new_var)>{var}
<any(new_var)><‘.property{var}’>
```

- **previousDot and dependentDot properties don’t modify <var>.**

When to make a copy of a Variable:

<nums.list>(5, 6, 7)
<temp.list><>

.insert,nis (1), (nums.list)<temp.list> **or** .insert,nis (1)<temp.list>{nums.list} § ~~<(7, 6, 5)>~~
Here, selfArgument or previousVar doesn’t need to make a copy of <nums.list>.

.insert,nis (1), (nums.list)<nums.list> § ~~<(7, 6, 5, 5, 6, 7)>~~
.insert. makes a copy of <nums.list> for selfArgument.

.len.

(e11)

*..len (1)<out>{names.list} generates **ASE**;*

*..len (-8:1)<out>{names.list} generates **AAE**;*
Here, (7-8+1 = 0) isn't between 1 and 7.

..len (3:-5)<out>{names.list} $ ~~(1)~~
Here, (3) and (7-5+1 = 3) both are same and between 1 and 7. So, It returns (1).

(e21)

..len (-2:12)<out>{name} $ ~~(13)~~
Here, (25-2+1 = 24) and (12) both are between 1 and 25. So, (24-12+1 = 13).

(e23)

<out><'.len (3:-1){pi_250}'> $ ~~(250)~~
Here, (3) and (252-1+1 = 252) both are between 1 and 252. So, (252-3+1 = 250).

(e31)

..len<out>{set.list} $ ~~(4)~~

(e8)

..len<out>{fruits.list} $ ~~(1)~~
..get apple ..len<out>{fruits.list} $ ~~(8)~~

.get. with str()

- **It always returns as valueResult.**

(e21)

.get (12:15)<name> § ~~John~~

..get (24:23)<out>{name} $ ~~82~~

*.get (12:15), (1)<name> generates **MAE**;*

..get (12:15) ..get (1)<out>{name} ***Here, Second .get. generates PME;***

..get (12:15); ..get (1)<out>{name} $ ~~J~~

Action of ; is valueResult to str() ~~John~~

.get (12:15) .remove h<name> ***Here, .remove. generates PME;***

.get (12:15); .remove h<name> § ~~Jon~~

if .get (12:15) .bool{name}:

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

*..find John ..get<out>{name} generates **PME**;*

*Boolean bundles .find John .get .bool{name}, .find Jon .get .bool{name} also generate **PME**;*

*Boolean bundle .get (12:15); .bool{name} generates **ADE**;*

Action of ; is valueResult to str(). ***It also makes boolResult inaccessible***

.find. .find,all. and .count. with str()

(e21)

..find n<out>{name} $ ~~(4)~~

..find,all n<out>{name} $ ~~<(4, 15)>~~

..find am<out>{name} $ ~~(5)~~

..find,all am<out>{name} $ ~~<(5, 20)>~~

*..find john<out>{name} generates **AAE**;*

..count 82<out>{name} $ ~~(0)~~

..get (15) ..find<out>{name} $ ~~(4)~~

..get (15) ..find,all<out>{name} $ ~~<(4, 15)>~~

..get (15) ..count<out>{name} $ ~~(2)~~

if .find John .bool{name}:

    ..get (it)<out>{name} $ ~~J~~

Boolean bundle .find Jon .bool{name} works as (false).

(e23)

..find \.14<out>{pi_250} $ ~~(2)~~

..count 1415<out>{pi_250} $ ~~(3)~~

..get (3:6) ..find,all<out>{pi_250} $ ~~<(3, 18, 78)>~~

(e21) Find last position of “n”.

..get (-1:1); ..find n<temp_pos>{name} **makes <temp_pos> as (11).**

Action of ; is valueResult to str() ~~*that is reversed.*~~

<out>(.len{name}-temp_pos+1)

*Here, ..get (-1:1) ..find n<out>{name} generates **MAE**;*

or ..find,all n; ..get (-1)<out>{name}

Action of ; is endResult to nums() ~~*<(4, 15)>*~~

or <out>{max(.find,all n{name})}

$ ~~(15)~~

<name><tenet>

if .get (-1:1) .find .bool{name}:

    <out><Given sentence\/ word is readable from backwards as well.>

$ ~~Given sentence/ word is readable from backwards as well.~~

.get. with list()

<names.list><John>
<2d.list><<>>

..get (1), (1)<out>{names.list} generates MAE;
..get (1), (1)<out>{2d.list} generates AAE;

(e11, e12)
<couple.list><Sarah, Mosh>

..find Sarah ..get<out>{names.list} $ ~~Sarah~~
..find (couple.list) ..get (1)<out>{couples.list} $ ~~Sarah~~

<couples.list><<Sarah, Mosh>, <>, Mia>

.get (1)<couples.list> **makes it list() with 2 members.** § ~~<Sarah, Mosh>~~

.get (2)<couples.list> **makes it null.list().** § ~~<>~~

.get (3)<couples.list> **is actually packing.** § ~~<Mia>~~

.get (1) .remove Mosh<couples.list> **or** .get (1) .pop<couples.list>
.get (2) .append John<couples.list>
§ ~~<<Sarah>, <John>, Mia>~~

.get (3) .remove Mia<couples.list> generates ***PME****;*
.get (3) .pop<couples.list> generates ***PME****;*
.get (3) .append Bob<couples.list> generates ***PME****;*

.get (3); .remove Mia<couples.list> **is actually <couples.list>{none}**

Action of ; is valueResult to str() ~~Mia~~

§ ~~<>~~

.get (3); .remove a<couples.list> **is actually packing.**

§ ~~<Mi>~~

*.get (3); .pop<couples.list> generates **VPME**;*

*.get (3); .append Bob<couples.list> generates **VPME**;*

..get (1) ..remove Mosh<out>{couples.list} **or** ..get (1) ..pop<out>{couples.list}

$ ~~<Sarah>~~

<nums.list><>

```
if .get (1) .bool{nums.list} is false():
    .append (1)<nums.list>
```

§ ~~<(1)>~~

*Boolean bundles .get (2) .bool{nums.list}, .get (1:1) .bool{nums.list} generate **AAE**;*

(e11)

```
if .get (1:7) .bool{names.list}:
    <out>{it}
else:
    .append Bob<names.list>
    <out>{names.list}
```

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

(e12)

if .get (1), (-3) .bool{couples.list} is false():
.get (1) .insert (-3), (0)<couples.list>
§ ~~<<(0), Sarah, Mosh>, <John, Yara>, <Mia, Bob>>~~

Here, Boolean bundle .get (1) .get (-3) .bool{couples.list} also works as (false).
*But, Boolean bundle .get (1), (-4) .bool{couples.list} generates **AAE**;*

.find. .count. and .find,nis. with list()

(e11)

..find Sarah<out>{names.list} $ ~~(1)~~
..get (6) ..find<out>{names.list} $ ~~(1)~~
..get (1) ..count<out>{names.list} $ ~~(2)~~

*..find S<out>{names.list} generates **AAE**;*

*..get (1) ..count a<out>{names.list} generates **MAE**;*
..get (1); ..count a<out>{names.list} $ ~~(2)~~
Action of ; is valueResult to str() ~~Sarah~~ which has ~~two “a”s~~

<guests.list><<Lenin, M>, <Julia, F>, <>>
<null.list><>

*..find M<out>{guests.list} generates **ADE**;*
..count M<out>{guests.list} $ ~~(0)~~
..find (null.list)<out>{guests.list} $ ~~(3)~~

(e12)

*..get (1), (1) ..find<out>{couples.list} generates **ADE**;*

..get (1) ..get (1) ..find<out>{couples.list} $ ~~(1)~~

Here, second .get. receives namesVar ~~<Sarah, Mosh>~~ and gives valueResult ~~“Sarah”~~ to .find.

Which also receives namesVar ~~<Sarah, Mosh>~~

<couples.list><<Sarah, Mosh>, <>, Mia>

```
..find,nis (x.list)<out>{couples.list}
        // <x.list><Sarah, Mia, <>>
$ ~~<(0, 3, 2)>~~
```

.get. .find. .find,all. and .count. with dict()

(e1, e2, e3)

.get children<detail1.list> is actually packing. § ~~<(2)>~~
.get children, name<detail2.list> § ~~<John, Mia>~~
.get children, (1), ckey<detail3.list> § ~~<John, Jon>~~

(e4)

*..find India<out>{family_detail.list} generates **ADE**;*
..count India<out>{family_detail.list} $ ~~(0)~~

..find,all Sarah<out>{family_detail.list} $ ~~<(1, 6)>~~
..count Sarah<out>{family_detail.list} $ ~~(2)~~

*..get (1), age ..find<out>{family_detail.list} generates **ADE**;*
..get (6), name ..find<out>{family_detail.list} $ ~~(1)~~

..get (1) ..get age ..find<out>{family_detail.list} generates ADE;
..get (3) ..get name ..find<out>{family_detail.list} $ ~~(1)~~

In both examples, First .get. returns dict(=1). So, .find. either returns (1) or generates error.

(e4, e3)
<couple.list><Sarah, Mosh>
<all_names.list><John, Jon>

..find (couple.list)<out>{family_detail.list} generates ASE;

..get children ..find (all_names.list)<out>{detail3.list} generates ASE;

- **Here, meaning of Defined dict is such that both “John” and “Jon” shares same keys: name, age, status, wife, daughter.**
  So, ..get children ..find John<out>{detail3.list}, ..get children ..find Jon<out>{detail3.list} both examples give (1).

(e4)

*..find Sarah ..get ..pop<out>{family_detail.list} Here, .pop. generates **NAE**;*
..find Sarah ..get ..pop age<out>{family_detail.list} **pops key.**
$ ~~<name: Sarah>~~

*..find Sarah ..pop age<out>{family_detail.list} Here, .pop. generates **MAE**;*
..find Sarah ..pop; ..get (1)<out>{family_detail.list} **pops member.**
    Action of ; is endResult to dict(>1)
$ ~~<name: Mosh age: 52 status: Retired contact: (027)7041>~~

(e4, e3)

*..pop; ..get (.len{family_detail.list})<out>{family_detail.list} generates **AAE**;*
    Action of ; is endResult to dict(>1)
*.pop; .get (.len{family_detail.list})<family_detail.list> also generates **AAE**;*
    Action of ; is endResult to dict(>1)

- **In both examples, .pop. as previousDot returns dict() of length (6) and as simpleDot pops last member to make it of length (6). But selfArgument of .get. (.len{family_detail.list}) has already been dissolved as (7).**

.get children, (.get children .find Jon{detail3.list}), ckey<detail3.list>
<out>{detail3.list} $ ~~<John, Jon>~~

- **Here, Defined dict (of length (1) which has “children” key) is converted into Layer (of length (2) which doesn’t have it) by .get. as simpleDot. But selfArgument (.get children .find Jon{detail3.list}) has already been dissolved as (1).**

.get. followed by .find.

In case of .get. returning valueResult:

(e21)

..get (15) ..find<out>{name} $ ~~(4)~~

*..get (12:15) ..find J<out>{name} generates **MAE**;*

..get (12:15); ..find J<out>{name} $ ~~(1)~~

Action of ; is valueResult to str() ~~John~~

(e11)

..get (6) ..find<out>{names.list} $ ~~(1)~~

*..get (6) ..find a<out>{names.list} generates **MAE**;*

..get (6); ..find a<out>{names.list} $ ~~(2)~~

Action of ; is valueResult to name() ~~Sarah~~

In case of .get. returning list():

(e12)

..get (1) ..find Sarah<out>{couples.list} $ ~~(1)~~

*..get (1) ..find<out>{couples.list} generates **AAE**;*

In case of .get. returning dict(=1):

(e4)

..get (4) ..find Mia<out>{family_detail.list} $ ~~(1)~~

*..get (4) ..find India<out>{family_detail.list} generates **ADE**;*

Here, .find. either returns (1) or generates error.

.find. followed by .get. Vs .get. with selfArgument in case of dict(=1)

(e1, e2, e3)

- **.find. works on the basis of whether ckey is defined or not. So, even in case of dict(=1), .get. can get posArgument which is always (1).**

  ..find Sarah ..get husband_name<out>{detail1.list} ***Here, .find. generates ADE; (dict() is not defined.)***
  ..get husband_name<out>{detail1.list} $ ~~Mosh~~

  ..get husband ..find Mosh ..get name<out>{detail2.list} $ ~~Mosh~~
  ..get husband, name<out>{detail2.list} $ ~~Mosh~~

- **But .get. in case of dict(=1) always assumes that selfArgument is key.**

  *..get (1), husband_name<out>{detail1.list} generates **MAE**;*
  *..get (1)<out>{detail1.list} generates **AAE**;*

  *..get husband, (1), name<out>{detail2.list} generates **MAE**;*
  *..get husband, (1)<out>{detail2.list} generates **AAE**;*

  ..get (.find Sarah{detail3.list})<out>{detail3.list} generates AAE; (Here, selfArgument dissolved as (1).) ****

.find. returns ADE with or without boolResult

In case of list(), ADE with boolResult;
In case of dict(), if ckey is defined, ADE with boolResult;
Else, only ADE;

```
<couples.list><<Sarah, Mosh>, John>
<temp.list><Mia, Bob>

    if .find (temp.list) .bool<couples.list> is false():
        .append (temp.list)<couples.list> works.



<detail.list><name: Sarah>

    if .find Mosh .bool{detail.list} is false():  .find. generates ADE;
        <temp.list><'~~ckey(detail.list)~~': Mosh  status: Retired> generates DisE;
        .append (temp.list)<detail.list> is not ideal.

    ckey(detail.list : ckey)
    if ckey is not null() AND .find Mosh .bool{detail.list} is false():
        <temp.list><'ckey': Mosh  status: Retired>
        .append (temp.list)<detail.list> doesn't get implemented, doesn't show any error.
        So, this implementation is ideal.



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

    if .find Mosh .bool{detail.list} is false():
        <temp.list><'ckey(detail.list)': Mosh  status: Retired>
        .append (temp.list)<detail.list> works.

    ckey(detail.list : ckey)
    if ckey is not null() AND .find Mosh .bool{detail.list} is false():
        <temp.list><'ckey': Mosh  status: Retired>
        .append (temp.list)<detail.list> works.
```

.get. and .find. with tree()

(e5)

..find John ..get<out>{family_tree.list}
$ ~~<name: John age: 29 status: Married>~~
..get (3), name<out>{family_tree.list}
$ ~~John~~

"linked" returns dict(=1) or null.list().
Linked key if available, returns dict() with varying length. Else, generates AAE;

(e5)

..get (3), linked<out>{family_tree.list}
$ ~~<mother: Sarah father: Mosh sister: Mia wife: Yara daughter: Sarah>~~
..get (3), linked, daughter<out>{family_tree.list}
$ ~~Sarah~~
*..get (3), linked, daughter, name<out>{family_tree.list} generates **MAE**;*

..get (3), daughter<out>{family_tree.list} **is actually** ..get (6)<out>{family_tree.list}
$ ~~<name: Sarah age: 12 months speaks: (true) words: da, mama>~~
..get (3), daughter, name<out>{family_tree.list}
$ ~~Sarah~~

(e6)

..get (1), linked<out>{tree.list}
$ ~~<children: <John, Mia>>~~
..get (1), linked, children, (1)<out>{tree.list}
$ ~~John~~

..get (1), children<out>{tree.list}
$ ~~<name: John,~~
~~name: Mia>~~
..get (1), children, (1), name<out>{tree.list}
$ ~~John~~

(e7)

..get (1), linked<out>{fake_tree.list}
$ ~~<>~~

*..get (1), linked, husband<out>{fake_tree.list} generates **AAE**;*
Here, .get. with tree() handles “husband” selfArgument.
So, even if “linked” returned null.list(), this argument gets interpreted as key.

*..get (1), linked ..get ~~husband~~<out>{fake_tree.list} generates **ASE**;*
Here, .get. with null.list() handles “husband” selfArgument because first .get. returned null.list(). So, this argument gets interpreted as position.

Similarly, Boolean bundle .get (1), linked, husband .bool{fake_tree.list} **works as (false)**.
*But, Boolean bundle .get (1), linked .get ~~husband~~ .bool{fake_tree.list} generates **ASE**;*

*..get (1), husband<out>{fake_tree.list} generates **AAE**;*

Here, Boolean bundle .get (1), husband .bool{fake_tree.list} **works as (false)**.
*But, Boolean bundle .get (1), husband, name .bool{fake_tree.list} generates **MAE**;*

With tree(), .get. can’t get as keyArgument.

(e5)

..find John ..get sister ..fetch India ..get<out>{family_tree.list} $ ~~India~~
Here, first .get. gets (3) as posArgument with tree().
But, second .get. gets “residence” as keyArgument with dict()(=1), not with tree().

.get. with range() as selfArgument with list() or dict()

It gives members if (2:3), second member as it is if (2:2) and (reversed) members if (2:1).

(e11)

..get (3)<out>{names.list} $ ~~John~~

..get (3:4)<out>{names.list} $ ~~<John, Mia>~~
..get (3:4), (1)<out>{names.list} $ ~~John~~

..get (3:3)<out>{names.list} **or**
..get (3:-5)<out>{names.list} $ ~~John~~
*..get (3:3), (1)<out>{names.list} generates **MAE**;*

(e12)

..get (2)<out>{couples.list} $ ~~<John, Yara>~~
..get (2), (1)<out>{couples.list} $ ~~John~~

..get (2:-1), (1)<out>{couples.list} $ ~~<John, Yara>~~
..get (2:-1), (1), (1)<out>{couples.list} $ ~~John~~

..get (2:-2)<out>{couples.list} $ ~~<John, Yara>~~
..get (2:-2), (1)<out>{couples.list} $ ~~John~~
*..get (2:-2), (1), (1)<out>{couples.list} generates **MAE**;*

(e4)

..get (3)<out>{family_detail.list} **or**
..get (3:3)<out>{family_detail.list} **or**
..get (3:4), (1)<out>{family_detail.list} $ ~~<name: John birth_date: 20 Jan 1990>~~

..get (3:3), name<out>{family_detail.list} **or**
..get (3:4), (1), name<out>{family_detail.list} $ ~~John~~

*..get (3:3), (1), name<out>{family_detail.list} generates **MAE**;*
*..get (3:3), (1)<out>{family_detail.list} generates **AAE**;*

With list() or dict(), if .get. as simpleDot has range() as selfArgument at any position, it returns as endResult to prevent that segment being modified.
This rule doesn’t apply to .get. as dependentDot or previousDot.

(e11)

.get (-2) .remove,all<names.list>

Here, .get. gives valueResult ~~“Sarah”~~

§ ~~<Mosh, John, Mia, Yara, Bob>~~

.get (6:-1), (1) .remove,all<names.list>,

*.get (6:-1), (1) .remove,all a<names.list> Both statements generate **PME**;*

Here, .get. gives endResult ~~“Sarah”~~

<lst.list>(1, 2, (3), 4, 5)

.get (3:3) .append (-3)<lst.list> Here, .append. generates PME;

.get (3:3); .append (-3)<lst.list>

Action of ; is endResult to nums() ~~<(3)>~~

§ ~~<(3, -3)>~~

.get (3:1), (1) .append (-3)<lst.list> Here, .append. generates PME;

.get (3:1), (1); .append (-3)<lst.list>

Action of ; is endResult to nums() ~~<(3)>~~

§ ~~<(3, -3)>~~

.get (3:1) .get (1) .append (-3)<lst.list> Here, second .get. generates PME;

.get (3:1); .get (1) .append (-3)<lst.list>

Action of ; is endResult to nums() ~~<((3), 2, 1)>~~

§ ~~<((3, -3), 2, 1)>~~

.get (3:1); .get (1); .append (-3)<lst.list>

Action of second ; is segment-nums() to nums() ~~<(3)>~~

§ ~~<(3, -3)>~~

.get (3:1); .append (0); .get (1) .append (-3)<lst.list>

Action of second ; is endResult to nums() ~~<((3), 2, 1, 0)>~~

§ ~~<((3, -3), 2, 1, 0)>~~

.get (3:1); .get (1) .append (-3); .append (0)<lst.list>

Action of second ; is endResult of segment-nums() to nums() ~~<((3, -3), 2, 1)>~~

§ ~~<((3, -3), 2, 1, 0)>~~

<lst.list>(1, 2, (3, -3), 4, 5)

..get (3:3) ..remove (-3)<out>{lst.list} **or**

..get (3:1), (1) ..remove,all (-3)<out>{lst.list} **or**

..get (3:1) ..get (1) ..remove,pos (-1)<out>{lst.list} **works.**

$ ~~<(3)>~~

<names.list><Sarah, John, Yara, Mia, Bob>

.get (1:2); .append Mia<names.list> **or**

.get (1:4); .remove Yara<names.list>

§ ~~<Sarah, John, Mia>~~

..get (1:2) .append Mia<names.list>{names.list} generates MAE;

..get (1:4) ..remove Yara<out>{lst.list} **works.**

$ ~~<Sarah, John, Mia>~~

.get. .gather. .find. .count. and .find,all. with Layer

.gather. as previousDot and dependentDot, gives dict(=1) (*not Layer*) or null.list()
It as simpleDot generates PSE;

(e8)

..get apple, (3)<out>{fruits.list}
$ ~~<ckey: <Cameo, Jonagold> colour: Red pattern: streaks or spots pattern_colour: Yellow taste: sweet>~~

..get apple ..gather Cameo<out>{fruits.list}
$ ~~<colour: Red pattern: streaks or spots pattern_colour: Yellow taste: sweet>~~

..get mangoes ..gather Alphonso ..get known_as<out>{fruits.list}
$ ~~"King of mangoes"~~

(e3)

..get children ..count Jon<out>{detail3.list} $ ~~(1)~~

- **With Layer, .count. returns (0) or (1),**
  .find,all. returns null.list() or nums() with only one position.

(e8)

```
if .get apple .count Cameo{fruits.list} != 0:
        ..get apple ..find Cameo ..get ..pop ckey<out>{fruits.list}
or  if .get apple .find Cameo .bool{fruits.list}:
        ..get apple, (it) ..pop ckey<out>{fruits.list}
or  if .get apple .gather Cameo .bool{fruits.list}:
        <out>{it}
```

$ ~~<colour: Red pattern: streaks or spots pattern_colour: Yellow taste: sweet>~~

if .get apple, (1) .bool{fruits.list}:

..get apple, (1) ..pop ckey<out>{fruits.list}

Here, In both statements, .get apple, (1). gives Layer(=1)

or if .get apple .gather Granny Smith .bool{fruits.list}:

..get apple ..gather Granny Smith<out>{fruits.list}

In both statements, .get apple. gives Layer(>1)

$ ~~<colour: Bright Green skin: shiny stem: long and thin>~~

if .get apple .get (1), ckey .gather .bool{fruits.list}:

..get apple ..get (1), ckey ..gather<out>{fruits.list}

In both statements, .get apple. gives Layer(>1)

And .get (1), ckey. gives valueResult ~~Granny Smith~~ to .gather.

*Which on receiving valueResult generates **PME**;*

if .get apple, (1), ckey .gather .bool{fruits.list}:

..get apple, (1), ckey ..gather<out>{fruits.list}

*Here, Variable ~~<fruits.list>~~ is Simple dict(=1), **not Layer***

.get apple, (1), ckey. gives valueResult ~~Granny Smith~~ to .gather.

*But since .gather. doesn't have Layer as Variable, it generates **VPME**;*

Some Examples

(e4) Mia now lives in US.

```
.get (.find Mia{family_detail.list}) .change residence, US<family_detail.list>
```

*Here, .get (4) .change **resident**, US<family_detail.list> generates **AAE**;*

Add what Sarah speaks.

```
.get (6) .add words, da\, mama<family_detail.list>
```

*Here, .get (6) .add **speaks**, da|, mama<family_detail.list> generates **ADE**;*

Note: .find Sarah. gives you “Sarah” at first position.

How old is John?

```
import datetime

<out><He is ‘date.math(“y0”,
        date.current(),
        .get (3), birth_date{family_detail.list})’ years old.>
```

$ ~~He is 29 years old.~~ Here, date.current() has returned 18-12-2019.

Maybe Mia’s age is wrong. She is two years younger than John.

```
..find Mia<pos>{family_detail.list}
.get (pos) .change age, (date.math(“y0”,
        date.math(“y0”, date.current(), .find John .get birth_date{family_detail.list}),
        date(“y”, 2)))<family_detail.list>

..get (pos), age<out>{family_detail.list}
```

$ ~~27~~

(e5) Give John’s sister’s details

```
if .find John .get sister .bool{family_tree.list}:
    <out>{it}
else:
    <out><John has no sister…>
```

$ ~~<name: Mia age: 25 status: Married residence: India>~~

Is Bob close member?

```
if .get (1:6), (.find Bob{family_tree.list}) .bool{family_tree.list}:
    <out>{it}
else:
    <out><not a close member…>
```

$ ~~not a close member…~~

Here, .get. returns dict() with length (6) [**close members**] but selfArgument (find Bob{family_tree.list}) has been dissolved as (7).

Who is Sarah’s uncle?

```
..find,all Sarah<all_pos.list>{family_tree.list}

loop:
    : for pos in all_pos.list
    : if .get (pos), linked, uncle .bool{family_tree.list}:  is ideal
        <out>{it}
        collect(“f”)
```

$ ~~Bob~~

Here, Boolean bundle .get (pos), uncle, name .bool{family_tree.list} generates MAE; for “Sarah” at position (1). So, it is not ideal.

John writes about himself!

```
import datetime

<statement><My name is ‘.get (3), name{family_tree.list}’. I am ‘dt.design(.get (3), age{family_tree.list}, “%ua”)’ old.>
```

Here, ‘.get (3), age{family_tree.list}’ ‘sp(~~.get (3), age{family_tree.list}~~, “year”, “years”)’ generates ***FIE****;*

```
if .get (3), status{family_tree.list} is of(“M”) or of(“m”):
    <statement> ++ < My wife\, ‘.get (3), wife, name{family_tree.list}’ is ‘.get (3), wife, age{family_tree.list}’ years old.>

    if .get (3), linked, son .bool{family_tree.list}:
        <statement> ++ < My son\, ‘it’ is ‘.get (3), son, age{family_tree.list}’ old.>

    if .get (3), linked, daughter .bool{family_tree.list}:
        <statement> ++ < My daughter\, ‘it’ is ‘.get (3), daughter, age{family_tree.list}’ old.>
```

§ ~~My name is **John**. I am **29** years old. My wife, **Yara** is **25** years old. My daughter, **Sarah** is **12 months** old.~~

Note: Highlighted data was within variable. Underscored data came from function.

(e8) Gala is also used in salad.

..get apple ..gather Gala ..get use<use_of_gala>{fruits.list}
<use_of_gala> ++ < or making salad>
.get apple, (.get apple .find Gala{fruits.list}) .change use, (use_of_gala)<fruits.list>

..get apple ..gather Gala ..get use<out>{fruits.list}
$ ~~cooking or eating raw or making salad~~

Combine pattern, pattern_colour keys of Cameo

..get apple ..gather Cameo ..get pattern_colour<pattern_colour>{fruits.list}
..get apple ..gather Cameo ..get pattern<pattern>{fruits.list}
<pattern> ++ < with 'pattern_colour' colour>

..get apple ..find Cameo<pos>{fruits.list}
.get apple, (pos) .pop pattern_colour<fruits.list>
Here, .pop (pattern_colour). which is .pop Yellow. generates AAE;
.get apple, (pos) .change pattern, (pattern)<fruits.list>

..get apple ..gather Cameo ..get pattern<out>{fruits.list}
$ ~~streaks or spots with Yellow colour~~

Maybe Earligold variety of apple is erroneous. Change it with following variable:
<temp.list><ckey: Earligold ripen: early>

.get apple .get (1:7) .append (temp.list)<fruits.list> Here, .append. generates ***PME;***
.get (1:7). returns endResult ~~Layer(=7)~~

.get apple .get (1:7); .append (temp.list)<fruits.list> **or**
.get apple, (1:7); .append (temp.list)<fruits.list> **has loss.**
*Action of ; is endResult to Layer(**>1**)*

Statement makes Simple dict(=1) with keys apple & mangoes **into Layer(=8) with eight apple varieties *but loss of key "apple" & key-value pair "mangoes".***

Here, .get apple .pop<fruits.list>
.get apple .append (temp.list)<fruits.list> **works.**

Here, .get apple. gives segment-Layer(>1) to .pop. and .append.

Both give endResult ~~Layer(>1)~~ to Variable

So, Variable receives both endResult as well as segment-Layer
So, it updates only that segment with that endResult

Keep only 3 varieties each of keys apple & mangoes

.get apple, (1:3)<fruits.list> **has loss.** It makes Simple dict(=1) with keys apple & mangoes **into Layer(=3) with three apple varieties *but loss of key "apple" & key-value pair "mangoes".***

Here, .get apple .get (1:3)<fruits.list>
.get mangoes .get (1:3)<fruits.list> **works.**

Here, .get apple. and .get mangoes. give respective segment-Layer(>1) to .get (1:3).

Which gives endResult ~~trimmed Layer(=3)~~ to Variable

So, Variable receives both endResult as well as segment-Layer
So, it updates only that segment with that endResult

.get. as simpleDot

If Variable receives only valueResult/ endResult, it means you’re modifying entire Variable.

And if Variable receives both Segment and valueResult/ endResult, it updates only that Segment.

But if Variable receives Segment only, it means you’re reducing whole Variable to just that Segment.

Note: Operator ; makes Segment, valueResult, endResult *an individual Variable*.

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

.get (3) .append (-3)<nums.list> § ~~<(1, 2, (3, -3), 4)>~~

.get (3) .replace,num (1), (0-n)<nums.list> § ~~<(1, 2, (-3), 4)>~~

In both examples, .get (3). gives segment-nums() ~~<(3)>~~ to .append. and .replace,num.

Both give endResult ~~<(3, -3)>~~ and ~~<(-3)>~~ respectively to Variable

*Since Variable receives both segment-nums() and endResult, **it updates only that Segment***

.get (3)<nums.list> § ~~<(3)>~~

Here, Variable receives segment-nums() ~~<(3)>~~ only

It means you are reducing whole Variable to just that Segment

.get (1)<nums.list> **is packing.** § ~~<(1)>~~

.get (1) .remove,all<nums.list> § ~~<(2, (3), 4)>~~

In both examples, Variable receives valueResult ~~(1)~~ and endResult ~~<(2, (3), 4)>~~ respectively without any Segment

It means you are modifying entire Variable

<detail.list><name: Sarah children: <<John, Yara>, <Mia>>,
name: Mosh>

.get (1) .get children .get (1), (1)<detail.list>

*Here, .get (1). gives segment-dict(**=1**) ~~<name: Sarah children: <<John, Yara>, <Mia>>~~ to .get children.*

*Which gives segment-names(**2d**) ~~<<John, Yara>, <Mia>>~~ to .get (1), (1).*

Which gives valueResult ~~“John”~~ to Variable.

Variable also receives segment-names(2d) ~~<<John, Yara>, <Mia>>~~ from .get children.

*Which means Variable updates only that Segment with that valueResult **which is packing***

§ ~~<name: Sarah children: <John>,~~
~~name: Mosh>~~

.get (1) .get children .get (1) .get (1)<detail.list>

*Here, second .get (1). receives segment-names(**2d**) ~~<<John, Yara>, <Mia>>~~ AND gives segment-names(**1d**) ~~<John, Yara>~~ to last .get (1).*

Which gives valueResult ~~“John”~~ to Variable

Variable also receives segment-names(1d) ~~<John, Yara>~~ from second .get (1).

*So, you are updating that Segment with that valueResult **which is packing***

§ ~~<name: Sarah children: <<John>, <Mia>>,~~
~~name: Mosh>~~

.get (1) .get children .get (1) .get (1) .remove,all<detail.list>

*Here, .remove,all. receives segment-names(**1d**) ~~<John, Yara>~~ from second .get (1). and valueResult ~~“John”~~ from last .get (1). AND gives endResult ~~<Yara>~~ to Variable*

Variable also receives segment-names(1d) ~~<John, Yara>~~ from second .get (1).

So, you are updating that Segment with that endResult

§ ~~<name: Sarah children: <<Yara>, <Mia>>,~~
~~name: Mosh>~~

.get (1) .get children .get (2) .append Bob<detail.list>

*Here, .get (2). gives segment-names(**1d**) ~~<Mia>~~ to .append Bob.*

Which gives endResult ~~<Mia, Bob>~~ to Variable

Variable also receives segment-names(1d) ~~<Mia>~~ from .get (2).

So, you are updating that Segment with that endResult

§ ~~<name: Sarah children: <<John, Yara>, <Mia, Bob>>,~~
~~name: Mosh>~~

<names.list><<John>, Sarah>

<detail.list><ckey: John, ckey: Yara>

.get (1) .get (1); .remove h<names.list>

.get (1) .get ckey; .remove h<detail.list>

*In both examples, first .get. gives segment-names(1d) ~~<John>~~ and segment-Layer(**=1**) ~~<ckey: John>~~ respectively to second .get.*

Which gives valueResult ~~“John”~~ to ;

Action of ; is valueResult to str() ~~John~~

it also makes Segment inaccessible

.remove. receives str() ~~John~~ AND gives endResult ~~“Jon”~~ to Variable

Since Segment is inaccessible, Variable receives only endResult

*It means you are modifying entire Variable with that endResult **which is packing***

§ ~~<Jon>~~

.get (1) .replace,pos (1), Jon<names.list> works. § ~~<<Jon>, Sarah>~~

.get (1) .change ckey, Jon<detail.list> works. § ~~<ckey: Jon, ckey: Yara>~~

*Note: .get (1) .change ckey, Yara<detail.list> generates **ADE**;*

Note:

.get (1) .get (1); .remove John<names.list>

.get (1) .get ckey; .remove John<detail.list>

Here, .remove John. receives str() ~~John~~ given by ; AND gives endResult ~~(none)~~ to Variable

*So, Variable receives only endResult ~~(none)~~ **which is as <names.list>{none} and <detail.list>{none}***

§ ~~<>~~

<detail.list><name: John daughter: Sara, name: Yara>

.get (1) .pop daughter; .add daughter, Sarah<detail.list>

Here, .pop. receives segment-dict(=1) ~~<name: John daughter: Sara>~~ AND gives endResult ~~<name: John>~~ to ;

Action of ; is endResult to dict()
It also makes segment-dict() inaccessible

.add. receives dict() ~~<name: John>~~ AND gives endResult ~~<name: John daughter: Sarah>~~ to Variable

Since segment-dict() is inaccessible, Variable receives only endResult
It means you are modifying entire Variable with that endResult

§ ~~<name: John daughter: Sarah>~~
.get (1) .change daughter, Sarah<detail.list> works. § ~~<name: John daughter: Sarah, name: Yara>~~

<nums.list>(1, 2, (3), 4, 5)

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

Here, Variable receives endResult ~~<(1, 2, (3))>~~ without any segment

§ ~~<(1, 2, (3))>~~

.get (1:3); .append (6)<nums.list>

Action of ; is endResult ~~<(1, 2, (3))>~~ to nums()
.append. receives it AND gives ~~<(1, 2, (3), 6)>~~ as endResult

§ ~~<(1, 2, (3), 6)>~~

.get (1:3); .replace,num (1), (n-1)<nums.list>

.replace,num. receives nums() ~~<(1, 2, (3))>~~ AND gives ~~<(0, 2, (3))>~~ as endResult

§ ~~<(0, 2, (3))>~~

.get (3:3)<nums.list> **gets only third member which is nums().** § ~~<(3)>~~

Here, Variable receives endResult ~~<(3)>~~ without any segment

*.get (2:-4); .append (-2)<nums.list> **gets only second member which is num().** So, It generates **VarE**; (type num() doesn't support any property.)*

Action of ; is endResult ~~(2)~~ to num()

.get (3:1), (1)<nums.list>

Here, Variable receives endResult ~~<(3)>~~ without any segment

§ ~~<(3)>~~

*.get (3:1) .get (1)<nums.list> Here, second .get. generates **PME**;*

.get (3:1); .get (1)<nums.list>

Action of ; is endResult ~~<((3), 2, 1)>~~ to nums()
.get (1). gives segment-nums() ~~<(3)>~~ to Variable

Here, Variable receives segment-nums() ~~<(3)>~~ only
It means you are reducing whole Variable to just that Segment

§ ~~<(3)>~~

.get (3:1); .get (1); .append (-3)<nums.list>

.get (1). receives nums() ~~<((3), 2, 1)>~~ from first ; AND gives segment-nums() ~~<(3)>~~ to second ;

Action of second ; is segment-nums() to nums() ~~<(3)>~~

Here, .append (-3). receives nums() ~~<(3)>~~ AND gives endResult ~~<(3, -3)>~~ to Variable

So, Variable receives only endResult without any Segment

§ ~~<(3, -3)>~~

.get (3:1); .get (1) .append (-3); .append (0)<nums.list>

.get (1). gives segment-nums() ~~<(3)>~~ to .append (-3).

Which gives endResult ~~<(3, -3)>~~ to second ;

Second ; makes endResult to nums() ~~<(3, -3)>~~
it also makes segment-nums() inaccessible

Here, .append (0). receives nums() ~~<(3, -3)>~~ AND gives endResult ~~<(3, -3, 0)>~~ to Variable which receives no Segment

§ ~~<(3, -3, 0)>~~

- **.get (3:1); .append (0); .get (1) .append (-3)<nums.list> works.**

Here, .append (0). receives nums() ~~<((3), 2, 1)>~~ from first ; AND gives endResult ~~<((3), 2, 1, 0)>~~ to second ;

Action of second ; is endResult to nums() ~~<((3), 2, 1, 0)>~~

Here, .get (1). receives nums() ~~<((3), 2, 1, 0)>~~ AND gives segment-nums() ~~<(3)>~~ to .append (-3).

Which gives endResult ~~<(3, -3)>~~ to Variable

Variable also receives segment-nums() from .get (1).

So, you are updating that Segment ~~<(3)>~~ with that endResult ~~<(3, -3)>~~ with Variable being nums() ~~<((3), 2, 1, 0)>~~ from second ;

§ ~~<((3, -3), 2, 1, 0)>~~

(e4)

.get (7) .add residence, US; .get (4) .change residence, India<family_detail.list>

Here, .add. receives segment-dict(=1) AND gives endResult ~~<name: Bob residence: US>~~ to ;

*Action of ; is endResult to dict(**=1**)*
So, .get (4). generates AAE;

- **Here, two separate lines works:**
  .get (7) .add residence, US<family_detail.list>
  .get (4) .change residence, India<family_detail.list>

.get (7) .add residence, US; .rename name, id<family_detail.list>
*Here, Action of ; is endResult to dict(**=1**)* ~~<name: Bob residence: US>~~
So, .rename. works
§ ~~<id: Bob residence: US>~~

.replace. .replace,pos. .remove. .remove,pos. and .replace,all. with str()

<movie_name><twelve monkeys>

.replace twelve, 12<movie_name> **or**
<num>(12)
.replace twelve,(stringify(num))<movie_name> § ~~12 monkeys~~

*Here, .replace Twelve, (12)<movie_name> generates **ASE**;*
*.replace Twelve, 12<movie_name> generates **AAE**; (**without boolResult**)*

```
if .find Twelve .bool{movie_name} is false():
    .replace twelve, 12<movie_name>
else:
    .replace,pos (it:it+5), 12<movie_name>
```

§ ~~12 monkeys~~

..remove s<out>{movie_name} **or**
..remove,pos (-1)<out>{movie_name} $ ~~twelve monkey~~

..get (6) ..replace E<out>{movie_name}
..get (6) ..remove<out>{movie_name}
..find twelve ..replace,pos 12<out>{movie_name}
*All three statements generate **PME**; (**or gives “twElve monkeys”, “twlve monkeys”, “12welve monkeys” respectively.**)*
[Here, ..replace twelve, 12<out>{movie_name} works fine.]

..get (6) ..replace,all E<out>{movie_name} $ ~~twElvE monkEys~~

(e21)

..find John ..remove,pos<out>{name}

*..find John ..remove,pos (3)<out>{name} Both statements generate **PME**;*

*..find John; ..remove,pos (3)<out>{name} generates **VarE**; (type num() doesn't support any property.)*

Action of ; is posResult to num()

.replace. .remove. .replace,pos. .remove,pos. .replace,nis. and .remove,all. with list()

<nums.list>(4, 5, 6, ())

..remove (6)<out>{nums.list} $ ~~<(4, 5, ())>~~

<temp.list><>

*..replace (temp.list), (7)<out>{nums.list} generates **ASE**;*

*..replace,pos (4), (7)<out>{nums.list} generates **ADE**;*

*..find (temp.list) ..remove,pos<out>{nums.list} generates **ADE**;*

[Here, .pop (4)<nums.list>

.insert (4), (7)<nums.list> works.]

<names.list><Sarah, John, Mia>

<pos.list><'.find Sarah{names.list}', '.find Mia{names.list}'>

.sort<pos.list>

<new.list><Mia, Sarah>

.replace,nis (pos.list), (new.list)<names.list> § ~~<Mia, John, Sarah>~~

Note: Here, length of <pos.list>, <new.list> must be same.

<nums.list>(5, 4, 3, 2, 1)

..get (1:3) ..replace,pos (1), (1)<out>{nums.list}

Here, .get (1:3). gives nums()

or ..get (1:3); ..replace,pos (1), (1)<out>{nums.list}

Action of ; is nums() to nums()

or ..replace,pos (1), (1); ..get (1:3)<out>{nums.list}

Action of ; is endResult to nums()

$ ~~<(1, 4, 3)>~~

..get (1:3) ..remove (5)<out>{nums.list}

$ ~~<(4, 3)>~~

vs ..remove (5); ..get (1:3)<out>{nums.list}

$ ~~<(4, 3, 2)>~~

(e11)

.get (6) .replace Sara<names.list>

*..get (6) ..remove<out>{names.list} both statements generate **PME**; (**or modifies “Sarah” at first position instead.**)*

[Here, .replace,pos (6), Sara<names.list>
..remove,pos (6)<out>{names.list} works fine.]

..get (6) ..remove,all<out>{names.list}

$ ~~<Mosh, John, Mia, Yara, Bob>~~

..find Sarah ..remove,pos<out>{names.list}

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

vs ..remove,pos (6)<out>{names.list}

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

.replace,lst.

<lst.list><John, <Sarah>>

.pop (1)<lst.list>
.insert (1), (be.lst(“John”))<lst.list>
or .replace,lst (1)<lst.list> § ~~<<John>, <Sarah>>~~
Note: *.replace,lst (2)<lst.list> generates* ***AFE****;*

..find John ..replace,pos ~~(be.lst(“John”))~~<out>{lst.list} generates ***ASE****;*
..find John ..replace,lst<out>{lst.list} works. $ ~~<<John>, <Sarah>>~~
..find (be.lst(“Sarah”)) ..replace,lst<out>{lst.list} generates ***AFE****;*

<names.list><Sarah, Mosh, John>

.get (3)<names.list>
.append Yara<names.list> § ~~<John, Yara>~~

Here, .get (3); .append Yara<names.list> generates ***VPME****;*
Action of ; is valueResult to str()
And, .replace,lst (3); .append Yara<names.list> makes no sense. § ~~<Sarah, Mosh, <John>, Yara>~~
Action of ; is endResult to names()
And, .replace,lst (3); .get (3) .append Yara<names.list> isn’t useful here. § ~~<Sarah, Mosh, <John, Yara>>~~

.replace,lst (3); get (3); .append Yara<names.list> § ~~<John, Yara>~~
Action of second ; is segment-names() to names()

.replace,num.

<nums.list>(7, 9, 9)

Change (9) at third position:
.replace,num (3), (n-1)<nums.list> § ~~<(7, 9, 8)>~~
Is similar to .replace,pos (3), (.get (3){nums.list}-1)<nums.list>

Change (9) that is found first:

if .find (9) .bool{nums.list}:

.replace,num (it), (n-1)<nums.list> § ~~<(7, 8, 9)>~~

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

.get (3) .replace,num (1), (0-n)<nums.list> § ~~<(1, 2, (-3), 4)>~~

.get (4) .replace,num (n/2)<nums.list> generates ***PME****;*

.get (4); .replace,num (n/2)<nums.list> generates ***VarE****; (type num() doesn’t support any property.)*

Action of ; is valueResult to num()

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

.get (1) .replace,num (2), (n/2)<nums.list>

or if .get (1) .find (4) .bool{nums.list}:

.get (1) .replace,num (it), (n/2)<nums.list>

§ ~~<((1, 2), 3, ())>~~

if .get (1) .find (4) .bool{nums.list}:

.replace,num (it), (n/2)<nums.list> ***makes no sense.***

§ ~~<((1, 4), 1.5, ())>~~

.replace,num (3), (n+1)<nums.list> generates ***ADE****;*

.replace,num (4), (n+1)<nums.list> generates ***AAE****;*

<prices.list>(10000, 20000, 25000)

<out><prices are\: ‘temp.list’>

For ..replace,num (3), (n-1)<temp.list{prices.list}, $ ~~prices are: 10000, 20000, 24999~~

For ..lambda,new (a), (a-1)<temp.list{prices.list}, $ ~~prices are: 9999, 19999, 24999~~

.add. and .add,set.

For <lst.list><>,

.add name, Sarah<lst.list> § ~~<name: Sarah>~~
.add (1), one<lst.list> § ~~<(1): one>~~
.add,set (1), one<lst.list> § ~~<(1) : one>~~

For <lst.list><name: Sarah>,

.add age, (50)<lst.list> § ~~<name: Sarah age: (50)>~~
*.add,set age, (50)<lst.list> generates **VPME**;*

For <lst.list><(31) : Thirty One>,

.add Thirty Two, (32)<lst.list> **or**
.add,set Thirty Two, (32)<lst.list>
§ ~~<(31) : Thirty One,~~
~~(32) : Thirty Two>~~

For <example.list><**..**>,

```
if .len{example.list} <= 1:
    .add ckey, Mosh<example.list>
else:
    .get (1) .add ckey, Mosh<example.list>
```

For <example.list><>, § ~~<ckey: Mosh>~~
For <example.list><age: (52)>, § ~~<**ckey**: Mosh **age**: (52)>~~
*For <example.list><ckey: Sarah>, generates **ADE**;*

```
if .len{example.list} <= 1:
    .add age, (50)<example.list>
else:
    .get (1) .add age, (50)<example.list>
```

For <example.list><>, § ~~<age: (50)>~~

*For <example.list><age: (52)>, generates **ADE**;*

For <example.list><ckey: Sarah>, § ~~<ckey: Sarah age: (50)>~~

Properties can not merge or spread values across Layer:

<temp.list><John, Mia>

<example.list><ckey: ‘temp.list’>
§ ~~<ckey: John,~~
~~ckey: Mia>~~

<example.list><>
.add ckey, (temp.list)<example.list>
*§ ~~<ckey: <John, Mia>>~~ **doesn’t work properly.***

<temp.list><Sarah, Mosh>

<example.list><ckey: ‘temp.list’ residence: US>
§ ~~<ckey: John residence: US,~~
~~ckey: Mia residence: US>~~

<example.list><residence: US>
.add ckey, (temp.list)<example.list>
*§ ~~<ckey: <Sarah, Mosh> residence: US>~~ **doesn’t work properly.***

How to use .add. .add,set. with Segment:

For <example.list><<>>, <example.list><<>, John>,

.get (1) .add name, Sarah<example.list>
*.get (1) .add,set (1), One<example.list> Both generate **VPME**;*

For <example.list><key: <<>>>, <example.list><key: <<>, John>>,

.get key, (1) .add name, Sarah<example.list>

*.get key, (1) .add,set (1), One<example.list> Both generate **VPME**;*

For <example.list><key: <>>,

.get key .add name, Sarah<example.list>

§ ~~<key: <name: Sarah>>~~

.get key *.add,set (1), One*<example.list>

§ ~~<key: <(1) : One>>~~

For <example.list><name: Sarah, name: Mosh>,

.get (1) .add age, (50)<example.list> **works**

Here, .get (1). gives segment-dict(=1)

No need to check for Parent which is dict(>1)

For <example.list><key: <name: Sarah>>,

.get key .add age, (50)<example.list> **works**

Here, .get key. gives value-dict(=1)

No need to check for Parent which is dict(=1)

For <example.list><key: <name: Sarah, name: Mosh>>,

.get key, (1) .add age, (50)<example.list> **works**

Here, .get key, (1). gives value-dict(=1)

No need to check for Parent which is value-dict(>1)

For <example.list><ckey: Sarah, ckey: Mosh>,

*.get (1) .add ckey, Sarah<example.list> generates **AFE**; **(key “ckey” already exists.)***

For <example.list><age: (50), age: (52)>, <example.list><key: <age: (50), age: (52)>>,

.get (1) .add ckey, Sarah<example.list> ***or***

*.get key, (1) .add ckey, Sarah<example.list> Both generate **ADE**; **(Parent is dict(>1))***

Here, .get (1). gives segment-dict(=1) while .get key, (1). gives value-dict(=1)

Since you are adding "ckey", dict(>1), value-dict(>1) as a Parent generate error

For <example.list><key: <age: (50)>>,

.get key .add ckey, Sarah<example.list> **works**

Here, .get key. gives value-dict(=1)

Since you are adding "ckey", check for Parent which is dict(=1)

§ ~~<key: <ckey: Sarah age: (50)>>~~

.add,nis. and .add,keys.

Structure of selfArgument of .add. .add,nis. and .add,keys. :

	First selfArgument	Second selfArgument
.add.	A key (always num(), name())	A value
.add,nis.	keys (always list(1d))	same number of values (always list(1d/ 2d))
	dict (always dict(=1)	-
.add,keys.	keys (always list(1d))	Single common value

<fruit.list><ckey: Golden delicious colour: yellow>

<temp.list><ckey: Golden Delicious colour: Pale yellow or Cream stem: long and thin>
<temp1.list><ckey, colour, stem>
<temp2.list><Golden Delicious, Pale yellow or Cream, long and thin>

.pop ckey<fruit.list>
.add,nis (temp.list)<fruit.list> **or** .add,nis (temp1.list), (temp2.list)<fruit.list>
§ ~~<ckey: Golden Delicious colour: Pale yellow or Cream stem: long and thin>~~

.add,nis (temp.list)<fruit.list> **or** .add,nis (temp1.list), (temp2.list)<fruit.list>
§ ~~<ckey: <Golden delicious, Golden Delicious> colour: Pale yellow or Cream stem: long and thin>~~

<dct.list><>

<lst.list>(1, 2, 4)
.add,keys (lst.list), (true)<dct.list>
.add (3), (false)<dct.list>
§ ~~<(1): (true) (2): (true) (4): (true) (3): (false)>~~

.change. and .rename.

(e4)

*.get (1) .rename name, id<family_detail.list> generates **ADE**;*

```
..get (1)<temp.list>{family_detail.list}
.rename name, id<temp.list>
<out>{temp.list}  $ ~~<id: Sarah  age: 50>~~
```

Properties can modify alt_layer but can’t join two different members of Layer:

```
<example.list><ckey: <John, Yara>  daughter: Sarah>
        // alt_layers(“John”, “Josh”)
§ ~~<ckey: <John, Josh>  daughter: Sarah, ckey: Yara  daughter: Sarah>~~
```

*Here, changing “Josh” to “John” or “Yara” generates **ADE**; (value already exists in Layer)*

- **Change “Josh” to “Jon”**

```
<temp.list><John, Jon>
.get (1) .change ckey, (temp.list)<example.list>
```

or .get (1), ckey .replace,pos (2), Jon<example.list>

or .get (1), ckey .pop<example.list>

Note: Here, one more use of .pop. generates ADE; (Defined value can’t be null.list())

.get (1), ckey .append Jon<example.list>

> **Note:** .replace,pos. .pop. .append. can’t work on value that is base(). So, convert it into list() (**of only member**) using pack.alt_layer().
>
> ```
> <example.list><ckey: <John, Yara>  daughter: Sarah>
>         // alt_layers(“John”, none)
> § ~~<ckey: <John>  daughter: Sarah, ckey: Yara  daughter: Sarah>~~
> ```

Try to .rename. to “ckey” until you succeed:

<example.list><name: John daughter: Sarah,
name: Yara daughter: Sarah>
// pack_ckey(“name”)

*.get (1) .rename name, ckey<example.list> generates **ADE**; (Parent is dict(>1))*

.pop<example.list>
.rename name, ckey<example.list>
§ ~~<ckey: John daughter: Sarah>~~

*.get (1) .rename daughter, ckey<example.list> generates **ADE**; (Variable has already been defined.)*

del_ckey(example.list)
*.get (1) .rename daughter, ckey<example.list> generates **ADE**; (Parent is dict(>1))*

.pop<example.list>
.rename daughter, ckey<example.list>
§ ~~<ckey: Sarah name: John>~~

.rename name, father<example.list>
§ ~~<ckey: Sarah father: John>~~

<example.list><name: John daughter: <name: Sarah>>

*.rename daughter, ckey<example.list> generates **ADE**; (defined value can’t be dict())*

New concept: Instead of generating AFE, .add. overwrites:

```
<dct.list><name: Mia  residence: US>
```

Previously,

```
if .get residence .bool{dct.list}:
    .change residence, India<dct.list>
else:
    .add residence, India<dct.list>
or try:
    .add residence, India<dct.list>
except AFE;
```

Now,

```
.add residence, India<dct.list>
```

```
<index.list><(1): Sarah  (2): John  (3): Mia>
```

Previously,

```
if NOT .get (4) .bool{index.list}:
    .add (4), Mosh<index.list>
or try:
    .add (4), Mosh<index.list>
except AFE;
```

Now,

```
if NOT .get (4) .bool{index.list}:  Since old data is more important, use this condition to
prevent .add. from over-writing.
    .add (4), Mosh<index.list>
```

.fetch. and .fetch,all.

(e1, e2)

*..fetch John, son<out>{detail1.list} generates **MAE**;*
*..fetch John<out>{detail1.list} generates **AAE**;*
*..fetch John<out>{detail2.list} generates **ADE**;*

*..get children ..fetch John<out>{detail1.list} generates **MAE**;*
..get children ..fetch<out>{detail1.list} $ ~~children~~
..get children ..fetch John<out>{detail2.list} $ ~~name~~

*..get children, name, (1) ..fetch<out>{detail2.list} generates **ADE**;*
*..get children ..get name ..fetch<out>{detail2.list} generates **VPME**;*
Here, first .get. gives segment-dict(=1) to second .get.
Which gives segment-names() to .fetch. which generates error
..get children ..get name, (1) ..fetch<out>{detail2.list} $ ~~name~~

(e9)
<name><fellowship of ring>

<out><Movies released on same year as ‘name’ are\: ‘.fetch (name) .get .remove (name) {movies_release.list}’.>
Here, .fetch. gives num() ~~(2001)~~ as keyResult to .get.
Which gives names() to .remove.
Which gives names() as endResult to Variable
$ ~~Movies released on same year as fellowship of ring are: amèlie, sorcerer’s stone, ocean’s eleven.~~

(e5)
..get (3), linked ..fetch,all Sarah<out>{family_tree.list} $ ~~<mother, daughter>~~

(e31)
..fetch (33), (33)<out>{set.list} $ ~~Thirty three~~
..fetch (34), (34)<out>{set.list} $ ~~(34)~~

.bool.

(e21)

*..find John ..bool<statement>{name} generates **PSE**; (used as previousVar.)*

```
if .find John .bool{name}:
          <out>{true}
```

$ ~~(true)~~

```
<example.list><available: ‘true’>
```

```
if .get available .bool{example.list} is false():
          .add available, (true)<example.list>
```

.get available. returns true as boolResult and true as valueResult.

Here, .bool. gets true as boolArgument.

```
if .get available{example.list} is false():
          <out><out of stock…>
else:
          <out><stock is available…>
```

$ ~~stock is available…~~

Here, Variable gets true as valueArgument.

.range.

```
<nums.list><>
```

.range (1:3)<nums.list> **or** .range (1:3), (1)<nums.list>

*It generates **NameE** if nums.list doesn’t exist.*

*It generates **VPME** if nums.list is not null.list().*

or <nums.list>{ofRange(1, 3)}

or <nums.list>(1, 2, 3.0)

§ ~~<(1, 2, 3)>~~ **is range();**

<out>{interval(nums.list)} **$** ~~(1)~~

<nums.list><>

.range (1:1)<nums.list> **or** .append,nis (1:1)<nums.list>

§ ~~<(1)>~~ **is range();**

<out>{interval(nums.list)} $ ~~(0)~~

..get (1)<num>{nums.list} § ~~(1)~~

*For <num><>, .range (1:1)<num> generates **VPME**;*

*For <nums.list><>, ..range (1:1)<num>{nums.list} generates **PSE**;*

<nums.list><>

.range (1:1), (2)<nums.list> **or** .range (1:3), (3)<nums.list> **or** .append (1)<nums.list>

§ ~~<(1)>~~ **is not range();**

<out>{interval(nums.list)} $ ~~(0)~~

<nums.list><>

.range (3.5:-1.5), (3)<nums.list>

§ ~~<(3.5, 0.5)>~~ **is not range();**

<out>{interval(nums.list)} $ ~~(3)~~

*Here, .range (3.5:-1.5), ~~(-3)~~. generates **ASE**;*

Note: Function interval() always returns num(>=0).

<nums.list><>

.append (1:1)<nums.list>

§ ~~<((1))>~~ **Here, .get (1){nums.list} is range();**

<out>{interval(.get (1){nums.list})} $ ~~(0)~~

- **Thus, .append,nis (range). on null.list() is similar to**
    - .range (range).

- **.append (range). is similar to**
    - .append (nullList). **with**
      .get (nullList) .range (range).

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

.get (3) .range (3:7)<names.list> **or** .get (3) .append,nis (3:5)<names.list>
§ ~~<John, Yara, <(3, 4, 5, 6, 7)>>~~

.get (3) .range (3:7), (2)<names.list>
§ ~~<John, Yara, <(3, 5, 7)>>~~

.random. and .random,all.

<nums.list>(1:3)

.random,all<nums.list> § ~~<(1, 2, 3)>~~ **remains a range().**
*.random<nums.list> generates **PSE**;*

..random,all<new_nums.list>{nums.list} § ~~<(3, 1, 2)>~~
..random<num>{nums.list} § ~~(3)~~

<nums.list><>

*.range (1:3) .random,all<nums.list> generates **PME**;*
.range (1:3); .random,all<nums.list> § ~~<(1, 3, 2)>~~
Action of ; is endResult to nums()

For <num><**..**>,

.random (nums.list)<num> .random<num>{nums.list} .range (1:3); .random<num>
*For <num><>, generate **VPME**; (by first property.)*
*For <num>(0), generate **VarE**;*

***Note**: If Variable doesn't exist, they generate **NameE**;*

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

    For <new_nums.list><>, <new_nums.list>(4), (or If Variable doesn’t exist,)
        ..random<new_nums.list>{nums.list}  § ~~<(2)>~~ is packing.
        ..random,all<new_nums.list>{nums.list}  § ~~<(2, 1, 3)>~~

    For <new_nums.list>(4),
        ..random .append<new_nums.list>{nums.list} or
        .append (.random{nums.list})<new_nums.list>  § ~~<(4, 3)>~~
        ..random,all .append,nis<new_nums.list>{nums.list} or
        .append,nis (.random,all{nums.list})<new_nums.list>  § ~~<(4, 1, 3, 2)>~~


<lst.list><<Sarah, Mosh>, <>, (0), ‘true’>

    loop:
        : if .random{lst.list} is null.list():
            <out><You can\’t play anymore…>
            collect(“f”)
        else: ..
```

*Here, Use of .random,all. generates **PSE**;*

.shift. and .exchange.

<nums.list>(5, 2, 1, 5, 7, 1)

.shift (1), (4)<nums.list> § ~~<(2, 1, 5, **5**, 7, 1)>~~

.exchange (1), (4)<nums.list> § ~~<(**5**, 2, 1, **5**, 7, 1)>~~

(e11) Print about Mosh's family.

..find Mosh ..shift (1)<out>{names.list} $ ~~<Mosh, Sarah, John, Mia, Yara, Sarah, Bob>~~

Print Sarah and her children's names.

..find Mia ..exchange (2); ..get (1:3)<out>{names.list} $ ~~<Sarah, Mia, John>~~
Action of ; is endResult to names()

.sort. and .reverse.

<nums.list>(1, 9, 3, 8, 7, 4)

..get (1:5) ..sort<out>{nums.list} $ ~~<(1, 3, 7, 8, 9)>~~
..sort; ..get (1:5)<out>{nums.list} $ ~~<(1, 3, 4, 7, 8)>~~
Action of ; is endResult to nums()

(e11)
.reverse<names.list> § ~~<Bob, Sarah, Yara, Mia, John, Mosh, Sarah>~~

<nums.list>(3:5)
<names.list><John, Yara, <>>

.sort<nums.list> § ~~<(3, 4, 5)>~~ **remains a range().**
.get (3) .append,nis (.reverse{nums.list})<names.list> § ~~<John, Yara, <(5, 4, 3)>>~~ **Here, third member remains a range().**

.pop. and .pop,nis.

```
<nums.list>(7, 8, 9)
<couples.list><<Sarah, Mosh>>

    ..pop<new_nums.list>{nums.list}  § ~~<(7, 8)>~~
    ..get (-1)<popped_num>{nums.list}  § ~~(9)~~

    .pop<nums.list>  § ~~<(7, 8)>~~  Here, .remove,pos (-1). also works.
    .pop (1)<couples.list>  § ~~<>~~  Here, .remove,pos (1). generates ADE;

    loop:
        ; <2d.list><>
        : ..pop .append<2d.list>{nums.list}
        ..get (-1) .append<2d.list>{nums.list}
        .pop<nums.list>
    until:
        if nums.list is null():
            <out>{2d.list}
    $ ~~<((7, 8), 9, (7), 8, (), 7)>~~
```

(e11)

```
    ..find,all Sarah<all_pos.list>{names.list}  § ~~<(1, 6)>~~
    .pop,nis (.sort{all_pos.list})<names.list>  § ~~<Mosh, John, Mia, Yara, Bob>~~
    ..sort .~~pop,nis~~<names.list>{all_pos.list} generates PSE;

    ..find,all Sarah; ..sort ..~~pop,nis~~<out>{names.list} generates PME;
        Action of ; is endResult to nums()
    ..find,all Sarah; ..sort; ..~~pop,nis~~<out>{names.list} generates NAE;
        Action of second ; is endResult to nums()
```

(e3)

..get children ..get (-1)<mia_detail.list>{detail3.list}

Here, .get children. gives dict(>1)

.get children .pop<detail3.list>

Here, .get children. gives value-dict(>1)

.rename children, son<detail3.list>

.add daughter, (mia_detail.list)<detail3.list>

§ ~~<name: Sarah age: 50 residence: US~~
~~husband: <ckey: Mosh age: 52 status: Retired>~~
~~son: <ckey: <John, Jon> age: 29 status: Married wife: Yara daughter: Sarah>~~
~~daughter: <ckey: Mia age: 25 status: Married husband: Bob>>~~

*..get husband ..pop<out>{detail3.list} generates **NAE**;*

Here, .get husband. gives dict(=1)

*.get husband .pop<detail3.list> generates **NAE**;*

Here, .get husband. gives value-dict(=1)

*.get husband .pop (-1)<detail3.list> generates **AAE**;*

..get husband ..find Mosh ..pop<out>{detail3.list} $ ~~<>~~

Here, .pop. receives dict(=1) ~~details of key "husband"~~ as a Variable from .get. and ~~(1)~~ as a posArgument from .find.

(e8)

..get apple ..len<out>{fruits.list} $ ~~(8)~~

..get apple ..find Earligold ..pop; ..len<out>{fruits.list} $ ~~(7)~~

Here, .pop. receives dict(>1) ~~Varieties of apple~~ as a Variable and ~~(8)~~ as a posArgument AND gives endResult

Action of ; is endResult to dict(>1)

(e2)

.pop husband<detail2.list>
§ ~~<name: Sarah children: <name: <John, Mia> age_difference: 4> residence: US>~~
.get children .pop name<detail2.list> **pops ckey since it is dict(=1).**
§ ~~<name: Sarah children: <age_difference: 4> residence: US>~~

<keys.list><husband, children>
*.pop,nis (keys.list)<detail2.list> generates **VPME**;*

(e3)

*.get children, (1) .pop ckey<detail3.list> generates **ADE**; (because length of Layer is >1.)*

..get children, (1) ..pop ckey<out>{detail3.list} works.
$ ~~<age: 29 status: Married wife: Yara daughter: Sarah>~~

(e5, e6) Give Sarah's children's details.

..find Sarah ..get linked ..fetch Mosh ..pop<out>{family_tree.list}
Here, .pop. receives dict(=1) ~~Value of "linked" key~~ as a Variable from .get. and ~~husband~~ as a keyArgument from .fetch.
$ ~~<son: John daughter: Mia>~~

..find Sarah ..pop<out>{tree.list}
Here, .pop. receives tree() ~~<tree.list>~~ as a Variable and ~~(1)~~ as a posArgument
$ ~~<name: John parents: <name: <Sarah, Mosh>>,~~
~~name: Mia parents: <name: <Sarah, Mosh>> brother: <name: John>>~~

.append. .append,nis. .insert. and .insert,nis.

<names.list><Sarah>

<temp.list><Mosh>
.append (temp.list)<names.list> § ~~<Sarah, <Mosh>>~~
.append,nis (temp.list)<names.list> § ~~<Sarah, Mosh>~~

<temp><Mosh>
.append (temp)<names.list> § ~~<Sarah, Mosh>~~
*.append,nis (temp)<names.list> generates **ASE**;*

<couples.list><<John, Yara>>
*.append (couples.list)<names.list> generates **ASE**;*
.append,nis (couples.list)<names.list> § ~~<Sarah, <John, Yara>>~~

<n>([])
<num>(n)
<couples.list><<John, Yara>>
(e5)

*<var.list>{~~num~~} generates **VarE**;*
<var.list>{couples.list} § ~~<<John, Yara>>~~
<var.list>{family_tree.list} is tree()

For <var.list><>,
.append<var.list>{num}
.append<var.list>{couples.list}
*.append<var.list>{family_tree.list} all generate **ADE**;*

<names.list><Sarah, John, Mia>

.append (.get (1){names.list})<names.list> **or**

..get (1) .append<names.list>{names.list}

§ ~~<Sarah, John, Mia, Sarah>~~

Here, .get (1) .append<names.list> generates ***PME****;*

.get (1); .append<names.list> generates ***VPME****;*

Action of ; is valueResult ~~“Sarah”~~ to str()

<couples.list><<Sarah, Mosh>, <>>

.pop<couples.list>

..get (1) ..reverse .append<couples.list>{couples.list}

Here, .reverse. receives names() ~~<Sarah, Mosh>~~ and gives ~~<Mosh, Sarah>~~ as endResult to nextDot .append.

or

.get (2) .append (.get (1), (2){couples.list})<couples.list>

Here, .get (2). gives segment-null()

.get (2) .append (.get (1), (1){couples.list})<couples.list>

Here, .get (2). gives segment-names()

..get (1), (2) .get (2) .append<couples.list>{couples.list} Here, nextDot .get. generates PSE;

or

.get (2) .append,nis (.get (1) .reverse{couples.list})<couples.list>

or

.get (2) .insert,nis (1), (.get (1){couples.list})<couples.list>

§ ~~<<Sarah, Mosh>, <Mosh, Sarah>>~~

How to use them with Boolean bundle .get (pos) .bool{var} :

```
<nums.list>(5, 7, 1)

        if .get (4) .bool{nums.list} is false():
                .append (0)<nums.list>
                or .insert (4), (0)<nums.list>
                or .insert (-1), (0)<nums.list>
        § ~~<(5, 7, 1, 0)>~~
```

Here, Boolean bundle .get (-1) .bool{nums.list} works as (true).

```
        if .get (-4) .bool{nums.list} is false():
                .insert (-4), (0)<nums.list>
                or .insert (1), (0)<nums.list>
        § ~~<(0, 5, 7, 1)>~~


<null.list><>

        if .get (1) .bool{null.list} is false():
                .append (0)<null.list>
        or
        if .get (-1) .bool{null.list} is false():
                .insert (-1), (0)<null.list>
                or .insert (1), (0)<null.list>
        § ~~<(0)>~~
```

There is one missing number in nums() to make it range(). Fill it using .insert. :

<nums.list>(4, 5, 6) has (3) missing.

..find (4)<pos>{nums.list} § ~~(1)~~

.insert (pos), (3)<nums.list> § ~~<(3, 4, 5, 6)>~~

*Here, .insert (pos-1), (3)<nums.list> generates **ASE**;*

..find (4) .insert (3)<nums.list>{nums.list} § ~~<(4, 5, 1, 6)>~~ has no meaning.

Here, nextDot .insert. considers ~~posResult (1)~~ from previousDot .find. as a value and selfArgument ~~(3)~~ as a position.

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

..find (4)<pos>{nums.list} § ~~(3)~~

.insert (pos), (3)<nums.list> § ~~<(1, 2, 3, 4)>~~

Here, .insert (pos-1), (3)<nums.list> § ~~<(1, 3, 2, 4)>~~ has no meaning.

..find (2)<pos>{nums.list} § ~~(2)~~

.insert (pos+1), (3)<nums.list> § ~~<(1, 2, 3, 4)>~~

Here, .insert (pos), (3)<nums.list> § ~~<(1, 3, 2, 4)>~~ has no meaning.

..find (4) .insert (3)<nums.list>{nums.list} § ~~<(1, 2, 3, 4)>~~ works but it has no meaning.

Here, value ~~posResult (3) from previousDot .find.~~ and position ~~selfArgument (3)~~ are unusually same.

How to reverse a list using .insert,nis. :

<nums.list>(5, 6, 7)

<temp.list><>

.insert,nis (1), (nums.list)<temp.list> **or** .insert,nis (1)<temp.list>{nums.list}

<nums.list>{temp.list} del(temp.list)

§ ~~<(7, 6, 5)>~~

Here, .insert,nis (1), (nums.list)<nums.list> § ~~*<(7, 6, 5, 5, 6, 7)>*~~ *is unusual.*

Make <temp.list>(2, 3) as ((2, -2), 3). Using it make <nums.list>(1, 4) as (1, (2, -2), 3, 4) :

.pop (1)<temp.list>

<twos.list>(2, -2)

.insert (1), (twos.list)<temp.list>

Here, .insert,nis. makes no sense. § ~~***<(-2, 2, 3)>***~~

or

<null.list><>

.insert (1), (null.list)<temp.list>

.get (1) .append,nis (twos.list)<temp.list>

Here, .get. gives segment-null()

Here, .append. generates ADE;

.insert,nis (2), (.reverse{temp.list})<nums.list>

or .insert,nis (-2), (temp.list)<nums.list>

or ..reverse .insert,nis (2)<nums.list>{temp.list}

Here, .insert,nis. gets previousArgument

or .insert,nis (-2)<nums.list>{temp.list}

Here, .insert,nis. gets previousVar

.append. .insert. with dict()

<dict.list><name: John,
name: Yara>
<temp.list><name: Sarah>

*.get (1) .append (temp.list)<dict.list> generates **VPME**;*

Because .get. gives dict(=1) ~~<name: John>~~ with parent-dict(>1) ~~Variable <dict.list>~~

*.get (1:1) .append (temp.list)<dict.list> generates **PME**;*

.get (1:1); .append (temp.list)<dict.list> makes no sense since ~~details of "Yara"~~ has been lost.

Because action of ; is endResult to dict(=1) ~~<name: John>~~ and to make Parent inaccessible

.append (temp.list)<dict.list> works.

Note: .append. .insert. work with no Parent or Parent being dict(=1).

.append. .insert. with Layer

<layer.list><ckey: John,
    ckey: Yara>
    // alt_layers("John", none)
§ ~~<ckey: <John>, ckey: Yara>~~

    *.get (1), ckey .append Yara<layer.list> generates **ADE**;*

<layer.list><ckey: John,
    ckey: Mia,
    ckey: Sarah>
<temp.list><ckey: Sarah residence: US>

    .insert (1), (temp.list)<layer.list>
    § ~~<ckey: Sarah residence: US, ckey: John, ckey: Mia>~~

<layer.list><ckey: Sarah,
    ckey: John,
    ckey: Yara>
<temp.list><ckey: Sarah speaks: 'true'>

    .append (temp.list)<layer.list>
    § ~~<ckey: John, ckey: Yara, ckey: Sarah speaks: (true)>~~

pack(), join(), unpack() and similar actions by properties

How to join another list:

<names.list><Sarah, Mosh>
<temp.list><John, Mia, Yara>

Join at last:

join(names.list, temp.list : names.list)

or .append,nis (temp.list)<names.list>
Last member of selfArgument ~~"Yara"~~ at last position.

§ ~~<Sarah, Mosh, John, Mia, Yara>~~

Join at first:

join(temp.list, names.list : names.list)

or .insert,nis (1), (.reverse{temp.list})<names.list>
Last member of selfArgument ~~"John"~~ at (1)st position.

or <pos>(-.len{names.list}-1) § ~~(-3)~~
.insert,nis (pos), (temp.list)<names.list>
Last member of selfArgument ~~"Yara"~~ at (-3) ~ (new_len-3+1) ~ (5-3+1) ~ (3)rd position.

§ ~~<John, Mia, Yara, Sarah, Mosh>~~

(e11) Say something about Mosh's family.

<out><We are 'join("Mosh", .remove Mosh{names.list})'.>
$ ~~We are Mosh, Sarah, John, Mia, Yara, Sarah, Bob.>~~

join((0, 0+1), 1+1, &!”John” &true, “Mia” : lst.list)

or <lst.list><>
.append,nis (0:0+1)<lst.list> **or** .append,nis (0, 0+1)<lst.list>
Here, .append,nis (0), (0+1)<lst.list> generates ***MAE****;*

.append (1+1)<lst.list>

<temp.list><John, ‘true’>
.append,nis (temp.list)<lst.list>
Here, .append,nis (John, true)<lst.list> Here, “John” generates ***NameE****; while true generates* ***DisE****;*
Here, for <name><John>,
.append,nis (name, true)<lst.list>
.append,nis (name:true)<lst.list> Both generate ***DisE****;*

.append Mia<lst.list>

§ ~~<(0), (1), (2), John, (true), Mia>~~

pack((0:1), 2, &!”John” &true, “Mia” : 2d.list)

or <2d.list><>
.append (0:1)<2d.list>
Here, .append (0), (1)<2d.list> generates ***MAE****;*

.append (2)<2d.list>

<temp.list><John, ‘true’>
.append (temp.list)<2d.list>

.append Mia<2d.list>

§ ~~<<(0, 1)>, (2), <John, (true)>, Mia>~~

<num>(3)

pack(num : nums.list)

or join(num : nums.list)

or <nums.list><> .append (num)<nums.list>

or <nums.list>{num}

§ ~~<(3)>~~

<nums.list>(3)

unpack(nums.list : num)

or ..get (1)<num>{nums.list}

*Here, <~~num~~>{nums.list} generates **VarE**;*

§ ~~(3)~~

unpack(nums.list : pack(another_nums.list))

*Here, unpack(nums.list : another_nums.list) generates **FOE**;*

or pack(unpack(nums.list) : another_nums.list)

or <another_nums.list><> ..get (1) .append<another_nums.list>{nums.list}

or ..get (1)<another_nums.list>{nums.list}

or <another_nums.list>{nums.list}

§ ~~<(3)>~~

<any(name1)><Sarah, Mosh> is list().

<any(name2)><John, Yara> is list().

<any(name3)><Mia> is str().

pack(name1, name2, name3 : couples.list)

or <couples.list>{name3}

.insert (1), (name1)<couples.list>

.insert (2), (name2)<couples.list>

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

join(name1, name2, name3 : names.list)

or <names.list>{name3}
.insert,nis (1), (.reverse{name1})<names.list>
.insert,nis (-2), (name2)<names.list>

§ ~~<Sarah, Mosh, John, Yara, Mia>~~

unpack(couples.list : couple1.list, couple2.list, name)
<out>{couple2.list} $ ~~<John, Yara>~~

unpack(names.list : any(name1), any(name2), name3, name4, name5)
<out>{name3} $ ~~John~~
Here, unpack(names.list : ~~name1~~, ~~name2~~, name3, name4, name5) generates FOE;
(Variables <name1> and <name2> are still of p-list.) ***

<2d.list>((1, 2, 3))

unpack(2d.list : nums.list)
unpack(nums.list : num1, num2, num3)
or unpack(.get (1){2d.list} : num1, num2, num3)
<out>{num3} $ ~~(3)~~

join(2d.list, 4 : 2d.list) **is updater.**
or .append (4)<2d.list>
<out>{2d.list} $ ~~<((1, 2, 3), 4)>~~
*Here, pack(2d.list, 4 : 2d.list) generates **FIE**;*

<2d.list>((1, 2, 3), 4)

unpack(2d.list : temp.list, none)
pack(temp.list : 2d.list)

or pack(.get (1){2d.list} : 2d.list)
pack(unpack(2d.list) : 2d.list) § ~~<((1, 2, 3), 4)>~~ makes no change.
*unpack(2d.list : pack(2d.list), ~~none~~) generates **FOE**;*

or .pop<2d.list>
or .get (1)<2d.list>
Here, .get. gives segment-nums()

§ ~~<((1, 2, 3))>~~

<temp.list><>
<couple.list><<Sarah, Mosh>>
<names.list><John, Yara>
<name><Mia>

join(temp.list, couple.list, names.list, name : temp.list)

or .append (temp.list)<temp.list>
*Here, .append,nis (temp.list)<temp.list> generates **ASE**;*

.append,nis (couple.list)<temp.list>
.append,nis (names.list)<temp.list>
.append (name)<temp.list>

§ ~~<<>, <Sarah, Mosh>, John, Yara, Mia>~~
Here, join() gets four Variables and returns Variable with length (5).

join(), unpack() with dict()

- **pack() generates FIE with dict() (of any length).**

<name.list><name: John>
        // pack_ckey("name")
<temp.list><name: Yara>

        .append (temp.list)<name.list>
        or join(name.list, temp.list : name.list) works fine.

<name.list><name: John>
<temp.list><name: Yara>
        // pack_ckey("name")

        *.append (temp.list)<name.list> generates **ADE**;*
        ***or** join(name.list, temp.list : name.list) generates **FSE**; (ADE by .append.)*

<temp1.list><ckey: John age: (29),
        ckey: Mia age: (23)>
        // alt_layers("John", "Jon")

§ ~~<ckey: <John, Jon> age: (29),~~
~~ckey: Mia age: (23)>~~ **has length (2).**

<temp2.list><ckey: <John, Mia> status: Married,
        ckey: Jon wife: Yara daughter: Sarah,
        ckey: Mia husband: Bob>

§ ~~<ckey: John status: Married,~~
~~ckey: Mia status: Married husband: Bob,~~
~~ckey: Jon wife: Yara daughter: Sarah>~~ **has length (3).**

join(temp1.list, temp2.list : detail.list)

§ ~~<ckey: <John, Jon> age: (29) status: Married wife: Yara daughter: Sarah,~~
~~ckey: Mia age: (23) status: Married husband: Bob>~~ **with length (2).**

unpack(detail.list : temp1.list, temp2.list)

§ ~~<ckey: <John, Jon> age: (29) status: Married wife: Yara daughter: Sarah>~~
§ ~~<ckey: Mia age: (23) status: Married husband: Bob>~~ **Both are of length (1).**

(e8)

unpack(fruits.list : apples.list, mangoes.list)
*unpack(fruits.list : fruits.list, none) both generate **FIE**; (input can't be dict(=1).)*

..get apple<apples.list>{fruits.list}
.get apple<fruits.list> are not unpacking.

<layer.list> as
§ ~~<ckey: <John, Jon> status: Married,~~
~~ckey: Yara status: Married>~~

For <temp.list> as § ~~<ckey: <John, Yara> daughter: Sarah>~~,

*join(layer.list, temp.list : layer.list) generates **FSE**; (ADE by .append.)*

For <temp.list> as § ~~<ckey: John daughter: Sarah>~~,

join(layer.list, temp.list : layer.list)
§ ~~<ckey: Yara status: Married,~~
~~ckey: <John, Jon> status: Married daughter: Sarah>~~

.get,pair.

(e8)

.get pair apple<fruits.list>

or .get apple<fruits.list>

Here, .get. gives segment-dict(>1)

<fruits.list><apple: ‘fruits.list’>

Note: *<fruits.list><apple: ‘.get apple{fruits.list}’> is slow because of .get. being dependentDot.*

(e3)

.get husband .get,pair age<detail3.list> works.

Here, .get. gives value-dict(=1) of dict(=1)

*.get children, (1) .get,pair age<detail3.list> generates **ADE**; (because parent is defined dict(>1).)*

Here, .get. gives value-dict(=1) of value-dict(>1)

In both examples, .get,pair ckey. works fine.

or <temp.list><ckey: ‘.get husband, ckey{detail3.list}’>
.change husband, (temp.list)<detail3.list>

<temp.list><>
.add ckey, (.get children, (1), ckey{detail3.list})<temp.list>
Here, <temp.list><ckey: ‘.get children, (1), ckey{detail3.list}’> § ~~<ckey: John, Jon>~~ makes no sense as it has str() ~~“John, Jon”~~ as defined-value.

.get children .pop (1)<detail3.list>
.get children .insert (1), (temp.list)<detail3.list>

§ ~~<name: Sarah age: 50 residence: US husband: <ckey: Mosh> children: <ckey: <John, Jon>, ckey: Mia age: 25 status: Married husband: Bob>~~

(e5)

..find John ..get age<out>{family_tree.list} $ ~~29~~

..find John ..get ..get,pair age<out>{family_tree.list} $ ~~<age: 29>~~

Here, .get. gives dict(=1)

*Here, ..find John ..get,pair age<out>{family_tree.list} generates **PME**;*

..find John ..get wife<out>{family_tree.list} $ ~~<name: Yara age: 30>~~

..find John ..get ..get wife<out>{family_tree.list}

*..find John ..get ..get,pair wife<out>{family_tree.list} Both generate **AAE**; by second .get. and .get,pair. respectively.*

*In both examples, .get. gives dict(=1) **(without any linked keys)***

..get (3), linked, wife<out>{family_tree.list} $ ~~Yara~~

*..get (3) ..get linked, wife<out>{family_tree.list} generates **MAE** by second .get.*

*..get (3) ..get linked<out>{family_tree.list} generates **AAE** by second .get.*

*..get (3) ..get,pair linked<out>{family_tree.list} generates **ADE**;*

*In all examples, .get. gives dict(=1) **(without “linked” key)***

.lambda,new.

First selfArgument is a:

selfArguments	List1	List2
	(3)	**(3, 5, 7, 9)**
a, (a+2)~ (2+a) $$	(5)	(5, 7, 9, 11)
a, (a+a)~ (2*a) $$	(6)	(6, 10, 14, 18)
a, (a-min) $$$	(0)	(0, 2, 4, 6)
a, (max-a)~ (-a+max) $	(0)	(6, 4, 2, 0)
a, (a-x) $$	(0)	(0, 2, 4, 6)
a, (x-a)~ (-a+x)	(0)	(0, -2, -4, -6)
a, (3**a) $	(27)	(27, 243, 2187, 19683)
_a, (3**x) *****_	_(27)_	_(27, 27, 27, 27)_

$ means, implementation might be useful in real circumstances.

* means, implementation might corrupt data.

***** means, implementation generates error.

First selfArgument is x:

List (7) doesn’t use loop.

	(7, 1)	**(7, 2, 1, 5)**
x, (x+y) $	(8, 1)	(9, 3, 6, 5)
x, (y-x)~ (-x+y)	(-6, 1)	(-5, -1, 4, 5)
x, (x-min) $$	(6, 1)	(5, 1, 0, 5)
x, (x*max)	(49, 1)	(49, 4, 5, 5)
x, (max+x+y-min)~ (x+y+max-min)	(14, 1)	(14, 4, 10, 5)
*x, (y+max) ******	*(8, 1)*	*(9, 3, 10, 5)*

First selfArgument is y AND second selfArgument has z:

List (4, 3) doesn’t use loop.

	(4, 3, 9, 4, 5, 2)
y, (x+y+z+2)~ (2+x+y+z) $	(4, 18, 18, 20, 13, 2)
y, (x+y+z+y)~ (x+2*y+z) $	(4, 19, 25, 22, 16, 2)
y, (x+y-z) $$	(4, -2, 8, 8, 7, 2)
y, (x+2**y+z)	(4, 21, 519, 30, 38, 2)
*y, (x//y+z) ******	*(4, 10, 4, 7, 2, 2)*
*y, (y+z) ******	*(4, 12, 13, 9, 7, 2)*
*y, (x+y+z+min) ******	*(4, 19, 19, 22, 13, 2)*

****** *Instead of .lambda,new y, (y+z).,*

Use of .lambda,new x, (x+y). for the same list (4, 3, 9, 4, 5, 2) gives (7, 12, 13, 9, 7, 2).

****** *.lambda,new y, (x+y+z+min). assumes minimum of x, y, z.*

First selfArgument is y BUT second selfArgument doesn’t have z:
List (7) doesn’t use loop.

	(7, 1)	**(7, 2, 1, 5)**
y, (x+y) $	(7, 8)	(7, 9, 3, 6)
y, (x-y) $$	(7, 6)	(7, 5, 1, -4)
y, (y+min)	(7, 2)	(7, 4, 2, 6)
*y, (max+min) *******	*(7, 8)*	*(7, 9, 3, 6)*

****** .lambda,new y, (max+min). is similar to .lambda,new y, (x+y).*

.lambda,reduce.

First selfArgument is a:

	(7)	**(7, 2, 1, 9, 5)**
a, (a+a)~ *(2*a)* $	(7)	(24)
a, (a*a)~ *(a**2)* $$	(7)	(630)

Here, .lambda, reduce a, (a+a). does sum of all members. So, simplified second selfArgument (2*a) becomes wrong.

Similarly, .lambda, reduce a, (a*a). multiplies all members. So, simplified second selfArgument (a**2) has no meaning.

First selfArgument is xy:

List (7) doesn’t use loop.

	(7, 3)	**(7, 2, 1, 9, 5)**
xy, (x-y) $$	(4)	(5, -8, 5)
xy, (max-min) $$$	(4)	(5, 8, 5)
xy, (x//y)	(2)	(3, 0, 5)
xy, (max//min) $$$	(2)	(3, 9, 5)
xy, (x-min)	(4)	(5, 0, 5)
xy, (y+max)	(10)	(9, 18, 5)
xy, (x+y+max) *****	_(17)_	_(16, 19, 5)_
_xy, (2*x) *****_	_(14)_	_(14, 2, 5)_

First selfArgument is xyz:

List (4, 3) doesn’t use loop.

	(4, 3, 9, 4, 5, 2, 1, 2)
xyz, (x+y+z) $$	(16, 11, 1, 2)
xyz, (x*y*z) $	(108, 40, 1, 2)
xyz, (2+x+y+z)	(18, 13, 1, 2)
xyz, (x+2*y+z) **	(19, 16, 1, 2)
xyz, (x+y-z) $$$	(-2, 7, 1, 2)
xyz, (x+2**y+z) ****	(21, 38, 1, 2)

.lambda,filter.

First selfArgument is a:

		(3)	**(3, 5, 9, 1, 8, 3, 2, 7)**
a, max $		(3)	(9)
a, min $		(3)	(1)
a, (2*a>x**2) $$	(2*a>9)~ **(a>4.5)**	()	(5, 9, 8, 7)
a, (a//3=x) $$$$	(a//3=3)	()	(9)
a, (a>2+x)	(a>5)	()	(9, 8, 7)
a, (a>2*x) $$$	(a>6)	()	(9, 8, 7)
a, (a=x) $$	(a=3)	(3)	(3, 3)
a, (a+5<5) or (-a>0)~ (a<0)		()	()

.lambda,filter a, (a//3=x). on list with first member (3) filters all (9), (10), (11) found within list!
.lambda,filter a, (a//10=0). filters all (1-9) numbers found within list!

First selfArgument is x, y or xy:

List (3) doesn’t use loop.

	(3, 3)	**(1, 2, 3, 3, 3, 8, 8)**
x, max or (x>y)	(-, 3)	(-, -, -, -, -, -, 8)
x, (x=y)	(3, 3)	(-, -, 3, 3, -, 8, 8)
x, (x!=y) $$	(-, 3)	(1, 2, -, -, 3, -, 8)
x, (x=y-1) $$	(-, 3)	(1, 2, -, -, -, -, 8)
y, min	(3, -)	(1, -, -, -, -, -, -)
y, (y<=x)	(3, 3)	(1, -, -, 3, 3, -, 8)
y, (y=x)	(3, 3)	(1, -, -, 3, 3, -, 8)
y, (y=x+1) $$	(3, -)	(1, 2, 3, -, -, -, -)
xy, max	(3)	(2, 3, 8, 8)
xy, min	(3)	(1, 3, 3, 8)
xy, (x=y) or (y=x)	(3)	(3, 8)
xy, (x!=y)	()	(1, 3, 8)
xy, (y!=x)	()	(2, 8, 8)
xy, (x+1=y)~ (x=y-1)	()	(1, 8)
xy, (y-1=x)~ (y=x+1)	()	(2, 8)

For .lambda,filter x, max., .lambda,filter y, min, .., .., maximum means “>” and minimum means “<“ only.

But for .lambda,filter xy, max., .lambda,filter xyz, min, .., .., maximum means “>=” and minimum means “<=“.

***** ***.lambda,filter xy, (y-x=1). generates error;*** *(what to filter: x or y?)*

First selfArgument is xyz:

List (3, 5) doesn’t use loop.

	(3, 5, 9, 1, 8, 3, 2, 7)
xyz, max	(9, 8, 2, 7)
xyz, min	(3, 1, 2, 7)

.lambda,spread.

First selfArgument is x or y:

List (7) doesn’t use loop.

	(7, 1)	**(7, 1, 2, 2, 5)**
x, max or x, (y>x) $$	(7, 1)	(7, 2, 2, 5, 5)
x, min $$	(1, 1)	(1, 1, 2, 2, 5)
x, (y<=x)	(1, 1)	(1, 1, 2, 2, 5)
x, (2*y<x)	(1, 1)	(1, 1, 2, 2, 5)
y, max	(7, 7)	(7, 7, 2, 2, 5)
y, (x>=y)	(7, 7)	(7, 7, 2, 2, 5)
y, min or y, (x<y)	(7, 1)	(7, 1, 1, 2, 2)

Here, .lambda,spread x, min. and .lambda,spread x, (y<=x). are different. But they will always give you same result (1, 1, **2**, 2, 5). Yet for first example, third member is **same old** third member, while for second example, it is **updated** from fourth member.

Here, .lambda,spread x, (y=x). and .lambda,spread y, (x=y) makes no change to list.

(e21)

import datetime

<**name**><My name is John.>
<name> ++ < I am ‘date(“y”, 28)’.>

M	01	-25	h	14	-12
y	02	-24	n	15	-11
	03	-23	.	16	-10
n	04	-22		17	-09
a	05	-21	I	18	-08
m	06	-20		19	-07
e	07	-19	a	20	-06
	08	-18	m	21	-05
i	09	-17		22	-04
s	10	-16	2	23	-03
	11	-15	8	24	-02
J	12	-14	.	25	-01
o	13	-13			

(e22) <**10pi**>(3.1415926535)

(e23)

ofpi(250 : 250pi)
stringify(250pi : **pi_250**)

(e11) <**names.list**><Sarah, Mosh, John, Mia, Yara, Sarah, Bob>

(e12) <**couples.list**><<Sarah, Mosh>, <John, Yara>, <Mia, Bob>>

(e31)

```
<set.list><(3) : Three,
(2) : Two,
One : (1),
Thirty three : (33)>
```

(e1) <**detail1.list**><name: Sarah husband_name: Mosh children: (2)>

(e2)

```
<detail2.list><name: Sarah
       husband: <name: Mosh  status: Retired>
       children: <name: <John, Mia>  age_difference: ‘date(“y”, 4)’>
       residence: US>

       // pack_ckey(“children”, “name”)
         pack_ckey(“husband”, “name”)
```

(e3)

```
<detail3.list><name: Sarah  age: ‘date(“y”, 50)’  residence: US
       husband: <ckey: Mosh  age: ‘date(“y”, 52)’  status: Retired>
       children: <ckey: John  age: ‘date(“y”, 29)’,
              ckey: Mia  age: ‘date(“y”, 25)’,
              ckey: <John, Mia>  status: Married,
              ckey: Jon  wife: Yara  daughter: Sarah,
              ckey: Mia  husband: Bob>>
```

```
        // pack_ckey(“name”)
          alt_layers(“children”, “John”, “Jon”)
```

```
..get children<out>{detail3.list}
$ <ckey: <John, Jon>  age: 29  status: Married  wife: Yara  daughter: Sarah,
ckey: Mia  age: 25  status: Married  husband: Bob>
```

For alt_layers(), first positional argument (here, “John”) must be key-value. Second positional argument (here, “Jon”) can be key-value or null.base().

alt_layers(“children”, “Mia”, “John”) makes “children” as below:
<ckey: <Mia, John> age: 25 status: Married husband: Bob,
ckey: Jon wife: Yara daughter: Sarah>

(e4)

```
<family_detail.list><name: Sarah  age: ‘date(“y”, 50)’,
        name: Mosh  age: ‘date(“y”, 52)’  status: Retired  contact: \(027\)7041,
        name: John  birth_date: ‘dt.design(date(20, 1, 1990), “%l %vd %e”)’,
        name: Mia  age: ‘date(“y”, 25)’  residence: India,
        name: Yara  age: ‘date(“y”, 30)’,
        name: Sarah  speaks: ‘true’,
        name: Bob>

        // pack_ckey(“name”)

..get (3), birth_date<out>{family_detail.list}  $ 20 Jan 1990
```

(e5)

```
<key_config.list><husband : wife,
          son : mother,
          daughter : father,
          niece : uncle,
          sister : brother>

<family_tree.list><
name: Sarah  age: ‘date(“y”, 50)’ :
          husband: Mosh  son: John  daughter-mother: Mia,
name: Mosh  age: ‘date(“y”, 52)’ :
          son-father: John  daughter: Mia,
name: John  age: ‘date(“y”, 29)’  status: Married :
          sister: Mia  wife: Yara  daughter: Sarah,
name: Mia  age: ‘date(“y”, 25)’  status: Married  residence: India :
          sister_in_law: Yara  niece-‘none’: Sarah  husband: Bob,
name: Yara  age: ‘date(“y”, 30)’ :
          daughter-mother: Sarah  ‘none’: Bob,
name: Sarah  age: ‘dt.design(date(“m”, 12), “%ud”)’  speaks: ‘true’  words: da\, mama :
          uncle: Bob,
name: Bob>

          // pack_ckey(“name”)
            key_config(key_config.list)
```

Note: Use of **<none>** as a keyword argument.

syntax of easy structure of tree():

Member	**Continued links**	**New/ Additional links**
Sarah		Mosh, John, Mia
Mosh	John, Mia	
John	Mia	Yara, Sarah
Mia	Yara, Sarah	Bob
Yara	Sarah, Bob	
Sarah	Bob	
Bob		

Each member has some continued and some additional links to next members.
For example, First member "Sarah" starts with three new links: "Mosh", "John", "Mia".
For "Mosh", "John" and "Mia" are continued links. There is no additional link.
For "John", "Mia" is continued link and "Yara" and "Sarah" are additional links.

dissolving of key config:

name: Sarah **...** : husband-wife: Mosh son-mother: John daughter-mother: Mia,
name: Mosh **...** : son-father: John daughter-father: Mia,
name: John **...** : sister-brother: Mia wife-husband: Yara daughter-father: Sarah,
name: Mia **...** : **sister_in_law**: Yara **niece**: Sarah husband-wife: Bob,
name: Yara **...** : daughter-mother: Sarah **'none'**: Bob,
name: Sarah **...** : uncle-niece: Bob,
name: Bob

Note: **Single keys** are highlighted.

dissolving of easy structure of tree():

name: Sarah **...** linked: <husband: Mosh son: John daughter: Mia>,

name: Mosh **...** linked: <wife: Sarah son: John daughter: Mia>,

name: John **...** linked: <mother: Sarah father: Mosh sister: Mia wife: Yara daughter: Sarah>,

name: Mia **...** linked: <mother: Sarah father: Mosh brother: John sister_in_law: Yara niece: Sarah husband: Bob>,

name: Yara **...** linked: <husband: John daughter: Sarah>,

name: Sarah **...** linked: <father: John mother: Yara uncle: Bob>,

name: Bob **...** linked: <wife: Mia niece: Sarah>

syntax after dissolving easy structure of tree():

Member with position	"linked" key's links to other members
Sarah (1)	2, 3, 4
Mosh (2)	1, 3, 4
John (3)	1, 2, 4, 5, 6
Mia (4)	1, 2, 3, 5, 6, 7
Yara (5)	3, 6
Sarah (6)	3, 5, 7
Bob (7)	4, 6

This syntax shows where to go in order to find linked member.
For example, "Sarah" related to "Yara" is **sixth member**, not the first one.

(e6)

<**tree.list**><name: <Sarah, Mosh> : children-parents: <John, Mia>,
    name: John : **‘none’**-brother: Mia,
    name: Mia>

    // pack_ckey(“name”)

name: <Sarah, Mosh> linked: <children: <John, Mia>>,
name: John linked: <parents: <Sarah, Mosh>>,
name: Mia linked: <parents: <Sarah, Mosh> brother: John>

Sarah, Mosh (1, 0)	John, Mia	2, 3
John (2)	Mia	1, 0
Mia (3)		1, 0, 2

(e7)

<**fake_tree.list**><name: Sarah age: (50) : **‘none’**: Mosh,
    name: Mosh>

    // pack_ckey(“name”)

name: Sarah age: (50) linked: <>,
name: Mosh linked: <>

Sarah (1)	Mosh	-
Mosh (2)		-

(e8)

```
<fruits.list><
apple: <ckey: Granny Smith  colour: Bright Green  skin: shiny,
          ckey: Golden delicious  colour: Pale yellow or cream,
          ckey: Cameo  colour: Red  pattern: streaks or spots  pattern_colour: Yellow,
          ckey: Warcester  eye: shallow  surrounding: beaded,
          ckey: <Gala, Red Delicious>  eye: shallow  surrounding: bumpy,
          ckey: <Golden Delicious, Granny Smith>  stem: long and thin,
          ckey: Gala  size: tall and large  use: cooking or eating raw,
          ckey: <Fuji, Gala, Cameo>  taste: sweet>
mangoes: <ckey: Earligold,
          ckey: Alphonso  known_as: \”King of mangoes\”,
          ckey: Kesar  grows_at: Saurashtra\, Gujarat,
          ckey: Langra  grows_at: Uttar Pradesh,
          ckey: Raspuri  known_as: Queen of mangoes  shape: oval,
          ckey: Rajapuri  size: Large>>

          // alt_layers(“apple”, “Cameo”, “Jonagold”)
          alt_layers(“mangoes”, “Kesar”, “Gir Kesar”)


..get apple ..gather Golden delicious<dict.list>{fruits.list} <dict.list> is Simple dict(=1).
..get apple ..find Golden delicious<pos>{fruits.list}
.get apple .pop (pos)<fruits.list>

..get apple ..find Golden Delicious ..get<temp.list>{fruits.list} <temp.list> is Layer(=1).
.get apple .pop (.get apple .find Golden Delicious{fruits.list})<fruits.list>

loop:
          : for any(value) in temp.list AND key in keys
          : .add (key), (value)<dict.list>


.get apple .insert (pos), (dict.list)<fruits.list>
```

Here, ..get apple, (2)<dict.list>{fruits.list}
.get apple .pop (2)<fruits.list>
.change ckey, Golden Delicious<dict.list>
.get apple .insert (2), (dict.list)<fruits.list> make "apple", (2) as below:
<ckey: Golden Delicious stem: long and thin colour: Pale yellow or cream> which is unusual.

```
if .get apple .gather Earligold .bool{fruits.list} is false():
        <dict.list><ckey: Earligold  ripen: early>
        .get apple .append (dict.list)<fruits.list>

if .get mangoes .find Earligold .bool{fruits.list}:
        .get mangoes, (it) .add ripen, early<fruits.list>
```

```
..get apple<out>{fruits.list}
$ <ckey: Granny Smith  colour: Bright Green  skin: shiny  stem: long and thin,
ckey: Golden Delicious  colour: Pale yellow or cream  stem: long and thin,
ckey: <Cameo, Jonagold>  colour: Red  pattern: streaks or spots  pattern_colour: Yellow  taste:
sweet,
ckey: Warcester  eye: shallow  surrounding: beaded,
ckey: Gala  eye: shallow  surrounding: bumpy  size: tall and large  use: cooking or eating raw
taste: sweet,
ckey: Red Delicious  eye: shallow  surrounding: bumpy,
ckey: Fuji  taste: sweet
ckey: Earligold  ripen: early>
```

```
..get mangoes<out>{fruits.list}
$ <ckey: Earligold  ripen: early,
ckey: Alphonso  known_as: ″King of mangoes″,
ckey: <Kesar, Gir Kesar>  grows_at: Saurashtra, Gujarat,
ckey: Langra  grows_at: Uttar Pradesh,
ckey: Raspuri  known_as: Queen of mangoes  shape: oval,
ckey: Rajapuri  size: Large>
```

(e9)

<**movies_release.list**><
(2001): <fellowship of ring, amèlie, sorcerer\'s stone, ocean\'s eleven>
(2002): <two towers, city of god, pionist>
(2003): <return of king, curse of black pearl, kill bill 1>>

(e10)

<**movies_rating.list**><
(8.9): <<return of king, \>1.5m>>
(8.8): <<fellowship of ring, \>1.5m>>
(8.7): <<two towers, \>1.3m>>
(8.6): <<city of god, \>0.6m>, <pionist, \>0.6m>>
(8.3): <<amèlie, \>0.6m>>
(8): <<curse of black pearl, \>0.9m>, <kill bill 1, \>0.9m>>
(7.7): <<sorcerer\'s stone, \>0.5m>, <ocean\'s eleven, \>0.4m>>>