Functions

Introduction

stringify()
py(), type()

sp(), repeat(), in_words(), interval()
ofStr()
get_int(), get_float(), easy_read()
pad(), nums_pad()

ofRange(), ofLst(), ofDct(), ofTree(), ofSet()

last_set(), last_pair(), keys(), total_keys()
ckey(), update_ckey(), del_ckey(), rename_keys()

str_solve(), unsolved_strs(), math_operators(), compare_sides(), compare()

summary of easyA & main environment

env* means environment is needed each time you use its function.

parent*_() means all child-functions need this parent. Use _child() with this one.
Note: _*child() also works here.

*child() means this particular child needs its parent. Use parent() with this one.
Note: parent*_() has wrong interpretation here.

For inner.len(),
create inner*:
    def len();

For str_solve(),
create easyA:
    def str_solve();

For get_smpl(),
create easyA:
    def snip_():
        def _get_smpl();

For date.design(),
create datetime:
    def date_():
        def _*design();

import easyA (from easyA import *) **Note: * means everything.**
You can use entire environment.

import easyA, snip **or**
from easyA import snip (from easyA, snip import *)
You can use all functions under parent snip_().

from easyA import snip.get_str **or**
from easyA, snip import get_str
You can use get_str() only.

from datetime, date import design
You can use only one function date.design().
Note: Using this function as simply design() generates NameE;

def snip_():
    def _none(); **is imaginary function that returns (true) if input is Snip.**
    def _get_str();
    def _get_smpl();

import easyA, snip
Here, snip(), snip.get_str(), snip.get_smpl() all can be used.

from easyA, snip import none, get_str
Here, snip(), snip.get_str() both can be used.

from easyA import snip.none **or**
from easyA, snip import none
Here, Only snip() can be used.

stringify()

It takes unlimited str() or one Snip, num(), bool(), un(), 1d-list().

```
<n>([])  <num>(n+2)  <statement>(3>1)  <var.list><(0), John>

    str_solve(stringify(n), 1, num)
    <out><Statement is 'stringify(statement)'.>  $ ~~Statement is (true).~~

    <out><'var.list'.>  $ ~~0, John.~~
    <out><'stringify(&0 &true)'.>  $ ~~0, (true).~~
```

With other types, it returns string "(nullBase)", "(nullList)", "(2dList)", "(dict)", "(set)".

With one str(), it supports a character as stop_at.

```
<out>{stringify("My name is John", stop_at=space)}  $ ~~My |~~
```

While with many str(), it supports string as stop_at, in_between.

```
<out>{stringify("My", "name", "is", "John", stop_at=space)}  $ ~~MynameisJohn~~
<out>{stringify("My", "name", "is", "John", in_between=space)}  $ ~~My name is John~~
```

With 1d-list(), it supports string as stop_at, in_between.

```
<out>{stringify(&"My" &"name" &"is" &"John", stop_at="is")}  $ ~~My name is~~
<out>{stringify(&"My" &"name" &"is" &"John", in_between=space)}  $ ~~My name is John~~
```

py(), type()

Both are terminal functions.

<dct.list><ripen: early>
<out>{dct.list} $ ~~<ripen: early>~~
py(dct.list) $ ~~{“ripen”: “early”}~~

<out>{type(“John”)} $ ~~(str)~~

(e8) <out>{type(.get apple{fruits.list})} $ ~~(dict)~~ **has one argument with .get. being rfrDependent. (Because function is generator with get any(var); statement.)**
Vs <out>{type(fruits.list, “apple”)} $ ~~(dict)~~ **has more than one argument.**

sp()

It gets int()(>=0)
If it is (0) or (1), it is singular. Else, plural.

<out><you’sp(tries, “ “, “ still “)’have ‘tries’ more ‘sp(tries, “attempt”, “attempts”)’.>

For <tries>(3), $ ~~you still have 3 more attempts.~~
For <tries>(1), $ ~~you have 1 more attempt.~~
For <tries>(0), $ ~~you have 0 more attempt.~~

Here <out><you ‘sp(tries, “ “, “still”)’ have …> prints you have … for <tries>(1), <tries>(0).

sp(tries, none, “still”) generates FIE;
sp(tries, “”, “still”) generates SE;

<out><‘name’\’s ‘sp(.len{children.list}, “child is”, “children are”)’\: ‘children.list’.>

For <name><Sarah> <children.list><John, Mia>, $ ~~Sarah’s children are: John, Mia.~~
For <name><John> <children.list><Sarah>, $ ~~John’s child is: Sarah.~~

<temp.list><>
<out><'sp(.len{temp.list}, "is", "are")' '~~temp.list~~'> generates DisE;
Here, function works fine but String syntax doesn't allow type null.list().

- **Note:**
  - Book rating **is** 7.6.
  - Available weight **is** 7 kg.
  - Available units **are** 75.

repeat()

<i>(6)
<out>{repeat("fizz ", i/3)} $ ~~fizz fizz~~**(space)**

in_words()

import easyA

<price>(277)
<price>{in_words(price)} **or** <price>{in_words(price, lan="e")}
<price> ++ < only.>
§ ~~Two Hundred Seventy Seven only.~~

interval()

<out>{interval(lst.list)}
For <lst.list>(1), $ ~~(0)~~
For <lst.list>(1, 1.5), $ ~~(0.5)~~
For <lst.list>(1, 1.5, 2.5), $ ~~(a line generated by /b(null))~~

<out>{interval(lst.list, 0.5)}
For <lst.list>(1, 1.5), $ ~~(true)~~
For <lst.list>(1, 1.5, 2.5), $ ~~(false)~~

if nums.list is nums():
    if .len{nums.list} = 1: **It can be range(). Here, interval() always returns (0).**
        <out><Only number in given list is '.get (1){nums.list}'.>
    elif nums.list is range() OR interval(nums.list) is not null():
        ..get (1)<num1>{nums.list}
        ..get (-1)<num2>{nums.list}
        <out><This list has numbers from 'min(num1, num2)' to 'max(num1, num2)'.>

ofStr()

import datetime

<detail><I am ‘date(“y”, 28)’ years old.>
<str><abc**012**d3e**fgh**4567>

ofStr(detail, space : lst.list)
§ ~~<I, am, 28, years, old>~~ Here, third member is not Snip.

ofStr(detail, “ “, broken_nums= true : lst.list)
§ ~~<(28)>~~

ofStr(detail, “ “, broken_nums= false : lst.list)
§ ~~<>~~

ofStr(str, 3, broken_nums= true : nums.list)
§ ~~<(12, 456, 7)>~~

ofStr(str, 3, broken_nums= false : nums.list)
§ ~~<>~~

Notes:

Keyword argument **mixed as true** gives you names() with member num() wherever possible.
Its default false means you are getting fstr().

Keyword argument **broken_nums as true** makes sure that you get your nums(). So, it removes parts of string that can’t be converted into num().
But **as false**, if there is any part of string that can’t be converted into num(), it returns null.list() instead of nums().
Its default none means you are getting fstr().

<detail><My name is John. My wife is Yara.>

loop:
	; ofStr(detail, “My” : temp.list) § ~~< name is John. , wife is Yara.>~~
	; <temp><His>
	; copy(temp)
	: for detail in temp.list
	: <temp> += detail
	<out>{temp/db}
	reset(temp)
finally:
	<out><>

$ ~~His name is John. His wife is Yara.~~

get_int()

It gets num() and returns **rounded int()**. For it, it takes second argument which is int()(>0).

	(8.6)	(87.6)	(876.6)	(8765.6)
	(8)	(87)	(876)	(8765)
get_int(.., 1)	(8) instead of FIE	(80)	(870)	(8760)
get_int(.., 2)	FIE	FIE	(800)	(8700)
get_int(.., 3)	“	“	FIE	(8000)
get_int(.., 4)	“	“	“	FIE

Seller wants to offer discount of 1500/-. But he wants to apply fancy-looking price, if he can. That way customers won’t notice less discount! But even in this case, least amount of discount must be 1200/-. Less than that and he won’t be able to complete sale!

```
import easyA

<out><Original price\: ‘price\/- /db>
<temp_discount>(price - get_int(price, 4))

if in_range(temp_discount, 1200, 1500):
        <sale_price>(price - temp_discount - 1)
else:
        <sale_price>(get_int(price-1500, 2) - 1)

<out><Now only\: ‘sale_price\/->
```

For <price>(20377), (20377-20000=377) which is too less.
Here, (20377-1500=18877~18800)
$ ~~Original price: 20377/- Now only: 18799/-~~

For <price>(21377), (21377-20000=1377) which is appropriate.
So, (21377-1377=20000) **which is already fancier**.
$ ~~Original price: 21377/- Now only: 19999/-~~

For <price>(23377), (23377-20000=3377) which is too much.
Here, (21377-1500=21877~21800)
$ ~~Original price: 20377/- Now only: 21799/-~~

get_float()

It gets float() and returns **same or short float()**.

import easyA

get_float(num, 2 : num)
<out>{num}

For <num>(8765), generates FIE;
For <num>(8765.6), $ ~~(8765.6)~~

(e22)

<out>{get_float(10pi, 4)} $ ~~(3.1416)~~

easy_read()

It gets num() and returns **str() with commas placed for easy reading**. Where to put commas is decided by subsequent arguments that are all int()(>1).

import easyA

<out>{easy_read(price, 3)}
For <price>(109999), $ ~~109,999~~
For <price>(1099.99), $ ~~1,099.99~~

<out>{easy_read(price, 3, 2)}
For <price>(109999), $ ~~1,09,999~~

<out>{easy_read(199.01, 2, 2, 3)} $ ~~1,99.01~~

Note:
<out>{easy_read(199.01, 2, 1)}
<out>{easy_read(199.01, 2, 2, 1)} In both examples, int()(=1) generates FIE;

<nums.list>(99, 100, 101)

loop:
	; <temp.list><>
	: for num in nums.list
	: .append (easy_read(num, 2))<temp.list>

finally:
	<out>{temp.list}

$ ~~<99, 1,00, 1,01>~~

pad(), nums_pad()

pad() gets num() while nums_pad() gets nums(). Both returns padded str() for better alignment.

Note: nums_pad() returns only to outOperator.

import easyA

pad(num, none, none : str) **or** pad(num, 0, 0 : str) ***Note:*** *pad(num : str) generates **FIE;***
For <num>(11.5), <str><11.5>
Here, (none) means no padding. Here, (0) is also similar to (none).

<out>{pad(num, 1, 0)} ***Note:*** *<out>{pad(num, 1)} generates **FIE;***
For <num>(1), $ ~~1~~
Here, padding of (1) at start doesn't make any difference.

<out>{pad(num, none, 1)} **or** <out>{pad(num, 0, 1)}
For <num>(10), $ ~~10.0~~
For <num>(11.5), $ ~~11.5~~
If Variable is float(), padding of (1) after decimal doesn't make any difference.

<out>{pad(num, 2, 2)}
For <num>(1), $ ~~01.00~~
For <num>(10), $ ~~10.00~~
For <num>(11.5), $ ~~11.50~~

<out>{nums_pad(nums.list, 0, 0)}
For <nums.list>(1, 10, 115), $ ~~001, 010, 115~~
For <nums.list>(1, 10, 11.5), $ ~~01.0, 10.0, 11.5~~
Here, (0) means maximum padding.
By default, keyword argument break is false.

<nums.list>(1, 10, 115)
<out><nums are\: \(/db>
<out>{nums_pad(nums.list, 2, none)/db}
<out><\).>

$ ~~nums are: (01, 10, 115).~~
$ **(this new line is by /b(null).)**

<out><nums are\:/b 'nums_pad(nums.list, 2, none, break= true)'>
$ ~~nums are:~~ **now, /b(space) gets us a new line.**
$ ~~01~~ **new line by function.**
$ ~~10~~ **"**
$ ~~115~~ **now, /b(null) gets us a new line.**
$

ofRange()

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

loop:
    ; <dct.list><>
    ; ofRange(1, .len{names.list}/2, interval=0.5 : temp.list)
    : for num in temp.list
    : for name in names.list
    : .add (num), (name)<dct.list>
finally:
    parent(dct.list)

<out>{dct.list}
$ ~~<(1): Sarah (1.5): Mosh (2): John (2.5): Yara (3): Mia (3.5): Bob>~~

<out>{ofRange(9, 9.7, interval=0.1)}
$ ~~<(9, 9.1, 9.2, 9.3, 9.4, 9.5, 9.6, 9.7)>~~

ofLst(), ofDct()

<names.list><John, Yara>

ofLst(names.list, “ckey” : layer.list) makes Layer. Here, keyword argument defined is always true().
ofLst(names.list, “name”, defined= true : defined_dict.list) makes Defined dict.
Previous implementation for this used inner.addFirst(var, value, key) where <key> is str() with aA-zZ only.. Note: Even property .add. doesn’t use inner.addFirst() with three inputs!

ofLst(names.list, “name”, defined= false : dict.list) **or** ofLst(names.list, “name” : dict.list) makes Simple dict.

ofDct(defined_dict.list : names.list)
§ ~~<John, Yara>~~

ofDct(dict.list : names.list)
§ ~~<>~~

(e5)

ofDct(family_tree.list : names.list)
§ ~~<Sarah, Mosh, John, Mia, Yara, Sarah, Bob>~~

ofTree()

(e5)

ofTree(family_tree.list : family_detail.list)
It pops "linked" keys of each member and adds linked key-value pairs using table {"linked" key's links to other members}

ofSet()

import datetime

<set.list><January : 'date("mm", 1)',
February : 'date("mm", 2)'>

ofSet(set.list : 2d.list)
§ ~~<<February, 02>, <January, 01>>~~

Note: Unlike ofStr(), ofSet() preserves member types of set(). So, Snip remains Snip.

last_set()

It gets set() and returns last entry of it as list() with length (2).

(e31)
<any(last_entry)>{last_set(set.list)}
§ ~~<Thirty three, (33)>~~

last_pair()

It gets dict(=1) and returns last key-value pair of it as dict(=1).

(e3)
last_pair(.get children, (1){detail3.list} : dct.list)
§ ~~<daughter: Sarah>~~

<dct.list><ckey: John>
if dcts(last_pair(dct.list), dct.list): **Here, Function last_pair() returns Layer.**
    <out><Dictionary has one key-value pair.>

keys(), total_keys()

Both gets dict(=1)
keys() returns list() while total_keys() returns int(>=1).

(e1)
<out>{keys(detail1.list)} $ ~~<name, husband_name, children>~~
<out>{repeat("*", total_keys(detail1.list))} $ ~~(3)~~

ckey()

It is generator. It gets dict() and returns it's ckey as a str().

(e5) if ckey(family_tree.list) = "name"; gets implemented.
(e8) <out>{ckey(fruits.list)} $ ~~(a line generated by /b(null).)~~
(e2) ckey(.get children{detail2.list} : ckey) § ~~name~~
*Unlike pack_ckey(), ckey(detail2.list, "children" : ckey) generates **FIE**;*

update_ckey(), del_ckey(), rename_keys()

They are updaters only.

update_ckey() gets dict(of any length) (and maybe input(s) showing path) and positional argument (str() representing ckey (including "ckey")) and returns defined-dict().

(e2)

<detail2.list> has no ckey.
update_ckey(detail2.list, "name")

It's key "husband" has "name" as a ckey.
update_ckey(detail2.list, "husband", "ckey")
..get husband<out>{detail2.list} $ ~~*<ckey: Mosh status: Retired>*~~

It's key "children" has "name" as a ckey.
*update_ckey(detail2.list, "children", "ckey") generates **FIE**;*
***or** ~~*<ckey: John age_difference: 04 residence: US,*~~*
~~*ckey: Mia age_difference: 04 residence: US>*~~ **which makes no sense.***

(e4)

<family_detail.list> has "name" as a ckey.
*update_ckey(family_detail.list, "ckey") generates **FIE**;*
or first member becomes** ~~*<ckey: Sarah age: 50 speaks: (true)>*~~ **which makes no sense.

(e5)

*update_ckey(family_tree.list, “ckey”) generates **FIE**; (because tree() is not modifiable. tree() can’t be Layer.)*

del_ckey() gets defined-dict(of any length) that is not Layer (and maybe input(s) showing path) and returns non-defined dict().

(e2, e4, e8)

<detail2.list> has key “children” with “name” as a ckey and <family_detail.list> has “name” as a ckey. While <fruits.list> has key “apple” with “ckey” as a ckey.

*update_ckey(detail2.list, “children”, none), update_ckey(family_detail.list, none) generate **FIE**;*
Note: Similarly, pack_ckey() with none as a input generates FIE;

del_ckey(detail2.list, “children”), del_ckey(family_detail.list) works.
Now, “name” is just a key.

*del_ckey(fruits.list, “apple”) generates **FIE**;*

rename_keys() gets dict(of any length) (and maybe input(s) showing path) and two positional arguments (str() representing old (can be “ckey”) and new key (can’t be “ckey”)) and returns dict().

(e2, e4)

<detail2.list> doesn’t have ckey. While <family_detail.list> has “name” as a ckey.
.rename name, member<detail2.list> works because <detail2.list> is non-defined.
*.get (1) .rename name, member<family_detail.list> generates **ADE**;*
del_ckey(family_detail.list)
.get (1) .rename name, member<family_detail.list> works.

rename_keys(detail2.list, “name”, “member”), rename_keys(family_detail.list, “name”, “member”) make dictionary with “member” being just a key.

(e8)

“apple” key of <fruits.list> has “ckey” as a ckey.
update_ckey(fruits.list, “apple”, “variety”)
Now, it has “variety” as a ckey.
del_ckey(fruits.list, “apple”)
Now, “variety” is just a key.

rename_keys(fruits.list, “apple”, “ckey”, “variety”) also changed ckey “ckey” to “variety” which is just a key.

Summary:

(e1)

update_ckey(detail1.list, “name”)
update_ckey(detail1.list, “member”)
or
rename_keys(detail1.list, “name”, “member”) **or** .rename name, member<detail1.list>
update_ckey(detail1.list, “member”)
Now, <detail1.list> has “member” which is ckey.

str_solve()

It gets String (that represents str.un()) as first argument, num() (to solve str.un()) as second argument and any number of arguments that can be num.un(), num(), bool.un() or bool().
It returns num() and bool() unchanged and tries to return num.un() as num() and bool.un() as bool().

```
import easyA
<n>([])
<num1>(4*n+12) or <num1>(n)  <num1> *= 4  <num1> += 12
<num2>(2-n) or <num2>(2)  <num2> -= n
<num3>(n*2/n) or <num3>(2)  <num3> /= n/n
<statement1>(num1>num2)
<statement2>(num1+num2=3*n+14)
```

- § ~~(12+4*n)~~
- § ~~(2-n)~~
- **§ ~~(2)~~ remains num() throughout**
- § ~~(4*n>-10-n)~~
- **§ ~~(true)~~ remains bool() throughout**

```
del(n)
str_solve("n", 2, num1, num2, num3, statement1, statement2)
```

- § ~~(20)~~ is now num()
- § ~~(0)~~ is now num()
- **§ ~~(2)~~**
- § ~~(true)~~ is now bool()
- **§ ~~(true)~~**

If solved with (-2)

- § ~~(4)~~
- § ~~(4)~~
- **§ ~~(2)~~**
- § ~~(false)~~
- **§ ~~(true)~~**

unsolved_strs(), math_operators()

Both gets num.un().
First one returns fstr() with str.un() as string.
Second one returns fstr() or 2d-names() with math operators as string. Or it can also return null.list().

import easyA

<n>([])
<max>([])

<out>{unsolved_strs(num)}
For <num>((3+4.2)*max-2+(n-1)) ~ (-2+(-1+n)+7.2*max), $ ~~<n, max>~~
For <num>(max+0) ~ (max), $ ~~<max>~~
For <num>(0-n) ~ (-n), $ ~~<n>~~

<out>{math_operators(num)}
For <num>((3+4.2)*max-2+(n-1)) ~ (-2+(-1+n)+7.2*max), $ ~~<+, <+>, +, *>~~
For <num>((-3+4.2)//n) ~ (1.2//n), $ ~~<//>~~
For <num>(3+(0-7)*n) ~ (3-7*n), $ ~~<-, *>~~
*For <num>(3//(0-7)*n) ~ (-3//7*n) ~ (0) generates **FIE**;*
For <num>(7//(0-7)*n) ~ (-7//7*n) ~ (-n), $ ~~<>~~

compare_sides()

It gets bool.un().

import easyA

<n>([])
<num1>(4*n+12)
<num2>(2-n)
compare_sides(statement: main, rfr, sign, another_sign)

For <statement>(n!=3),

<out>{main} $ ~~(n)~~ is num.un(), **not str.un()**.

<out>{rfr} $ ~~(3)~~ is num().

<out>{sign} $ ~~>~~ is character.

<out>{another_sign} $ ~~<~~ is character.

For <statement>(num1>num2) ~ (12+4*n>2-n),

<out>{main} $ ~~(12+4*n)~~ is num.un().

<out>{rfr} $ ~~(2-n)~~ is num.un().

<out>{sign} $ ~~>~~ is character.

<out>{another_sign} $ **(a line by /b(null))**

Note: Since function doesn't get bool(), both variables <main> and <rfr> together can't be num().

compare()

It returns bool() or bool.un().

<num>(4)

if compare(num, double(2), "="):

<out><double of 2.>

$ ~~double of 2.~~

<out>{compare(3, 2, "<")}

$ ~~(false)~~ **Here, both numbers generate ">" which doesn't match.**

<n>([])

<num1>(n+0) **and** <num2>(n*1)

<out>{compare(num1, num2, "=", "<")}

$ ~~(true)~~ **Both generate "=" which matches.**

<num1>(4*n+12) **and** <num2>(2-n)

<out>{compare(num1, num2, "<", ">")}

$ ~~(12+4*n!=2-n)~~

easyA environment

	get_int() to modify num()
	get_float() to modify float()
	easy_read() generates str() with commas placed from num()
	pad()
	nums_pad() ***
	in_words() generates str() from num()
Convert to other types	ofStr() to make 1d-list() from name()
	ofRange() to make 1d-nums() from given Range
	ofLst() to make dict() from list()
	ofDct() to make list() from Defined dict, Layer, tree()
	ofTree() to pop “linked” keys and add linked keys
	ofSet() to make 2d-list() from set()
Check members of list()	fname() to make sure names() has no num(), bool(), bit.one()

	fstr(), fsnip(), fbool()
	fint(), ffloat()
	fnull()
snip_()	_get_smpl() returns one or more num()
	_get_str() returns relatable str()
For Unsolved	str_solve() to make num(), bool() from num.un(), bool.un() respectively
	unsolved_strs(), math_operators() for num.un()
	compare_sides() for bool.un()

*** function returns only to outOperator.

get_int(), get_float() can be called near_num(), near_float() respectively.

ofLst() can make Simple dict. But that is not reversible to list() by function ofDct(). Instead, it returns null.list().

ofDct() can generate 2d-list() if defined key's value is names() or nums(). That 2d-list() is reversible to dict() by ofLst().

fname(), fstr(), fsnip(), fbool() always returns (false) with nums(), null.list().
Similarly, fint(), ffloat() always returns (false) with names(), null.list().

Some functions from main environment

if in_range(month, 1, 12); **vs** if month >= 1 AND month <= 12;

- if in_range(10, 1, 10, involve_last= false); doesn’t get implemented.
- if in_range(rating, 1, 10, involve_last= false); gets implemented for both <rating>(9.9), <rating>(9.99)
- if in_range(rating, 1, 9.9) is bad design as it doesn’t work for <rating>(9.99)

if divided_by(n, 2); **vs** if n/2 is int();
if of_len(stringify(year), 4); **vs** if .len{year_str} = 4;
if compare(4, 4.0, “=“); **vs** if 4 = 4.0;