Radu BERINDEANU

RPPy – Reverse Polish Python POC (Proof Of Concept)

User Manual
ed. 1

Crap can work. Given enough thrust, pigs could fly, but that's not necessarily a good idea: you don't know where they're going to land, and it's dangerous to sit under them as they fly overhead

Al Viro

Table of Contents

Introduction

Reverse Polish Python, or **RPPy**, is an attempt to use the *concatenative paradigm* applied to classic Python. Generally speaking, most programming languages use the application of function to arguments, where you define a function with formal arguments and use it with concrete values for those arguments.

In a concatenative language, there are no formal arguments: instead, all functions, called *words*, use a single type of data, usually a stack, to get their input values, leaving the output values also here, for the next functions to get. For example (user input in **bold**):

In Python:

```
def meanval(a,b):     # formal parameters a,b
    return(a+b)/2

meanval(10,20)        # assignment of values for a,b
15.0                  # result
```

In RPPy:

```
meanval: + 2 / ; .    # no formal parameters, only operators:
                      # "+" adds the two top stack values, leaving sum
                      # 2 puts its own value on top of stack
                      # "/" divide sum by 2, leave quotient
                      # ";" means return, word's execution terminated
                      # (ignore the dot ".", it means terminating user input)
10 20 .               # putting values on the stack
[10,20]               # RPPy displays the stack: two values, with 20 on top
meanval .             # call the word meanval, arguments are ready to use
[15.0]                # RPPy displays the stack, result ready for next word
print .               # print needs an argument, which is already here
15.0                  # result printed
[]                    # RPPy displays the stack: empty, as all arguments
                      # are consumed by meanval and print
```

In fact, more shortly:

```
10 20 meanval print .
15.0
[]
```

Programming in *concatenative style* means simply *composing words* to define new ones:

```
mvprint: meanval print ; .
10 20 mvprint .
15.0
[]
```

And why Reverse Polish? RPN, or Reverse Polish Notation, is perfectly suited for the way concatenative programming works: always the operators follow their operands, 20 + 30 is 20 30 + in RPN; there is no need for parentheses, as (10+20)*5 is 10 20 + 5 * , or if stretching your imagination

5 10 20 + * (take pencil and paper and work out the stack effect...)
In Python you have *strict precedence* for operators (PEMDAS - parentheses, exponentiation, multiplication, division, add, subtract); in RPPy the order of operations is *entirely at your choice*, as expression evaluation is always done left to right.

But what we gain with this concatenative approach? Here's a quick answer:

- *point-free style programming*: no need for named parameters, your whole program is nothing but a sequence of literals (values being pushed on the stack) and words using them.

- *concision*: the only "glue" between words is whitespace, a line of code in RPPy may include several calls to words by mentioning only their name, no need for parameter names and assigning values.

- *factoring*: any common sequence of words found in multiple definitions can be factored out as a separate definition, which leads to shorter programs enhancing readability.

- *interactivity*: at the console, define a word, put some values on the stack, test it - **WYSIWYE** - **What You See Is What You Execute!** ; the stack is *always visible*, no need to insert "print(xxx)" everywhere as debugging aid. Once tested, use it in other definitions; integrate from simple to complex until arriving at the final words of your application, a process known as *incremental compiling*.

- *shortest development cycle*: Python, in interpret mode, is also highly interactive, but even the smallest program needs a *separate source code*, the so-called *script*, which has to be edited , tested, corrected at source level with an editor, re-tested again. On the other hand, RPPy **has no source code!** You create and test interactively your set of words which can be saved at any instant as compiled code, reloaded again and continued by adding new words, or even redefining old words until finished. No need for a separate source editing phase.

- *spreadsheet-like data handling*: as there is no difference between words defining data and words defining functions, saving your set of words means saving each time a *current snapshot* of your program, data included. As with spreadsheets, what is saved in the cells at session end will be retrieved at next loading; no need to save data in separate files.

OK, all that's wonderful ;););), but what we loose?

- *patience*: the learning curve is steep, as mostly we are used with infix notation, not postfix RPN; all has to be done backwards...

- *attention*: it is more difficult to program in concatenative mode, as you have to pay attention not only to the control flow of program's execution, but also to the *data flow on the stack*. Loosing focus on what's happening on the stack is very easy, with consequences Not-To-Be-Named-Here...

Countermeasure: short definitions, thoroughly tested before advancing to the next one; minimise stack shuffling; comment every definition with the associated stack effect. More on this in “Using RPPy”

As prerequisite, knowing at least programming at ground level in Python is necessary: RPPy uses basic functions and methods from Python, but accessed only in concatenative mode. Unfortunately, learning RPPy will not improve your skills at Python programming: no indentation, no formal parameters, no statements, no expressions, no OOP, and the control flow at program execution is totally different.
But what you could find useful is mastering Python's data types and associated operations, most of them being integrated in RPPy.

Compared to Python, in RPPy you dispose only of a tiny fraction (about 250) of the vast number of functions & methods available in Python land, but even with this reduced set a good deal of data handling is covered.

And the grand final question: what is RPPy good at?

- quick & dirty *small apps* as it’s highly interactive with short development cycle, but difficult to scale up
- banging your head on the wall (or screen, if you have a prehistoric 15" monitor in your basement) as you dig further and further in the postfix world...

Have fun! Because that's the only reason why RPPy was created…

And Why Proof Of Concept?

Because it’s only an attempt to define an unusual programming environment exposing a tiny fraction of classic Python functions and methods. If RPPy will prove a certain usefulness, further development could be envisaged. Who knows...

Installing RPPy

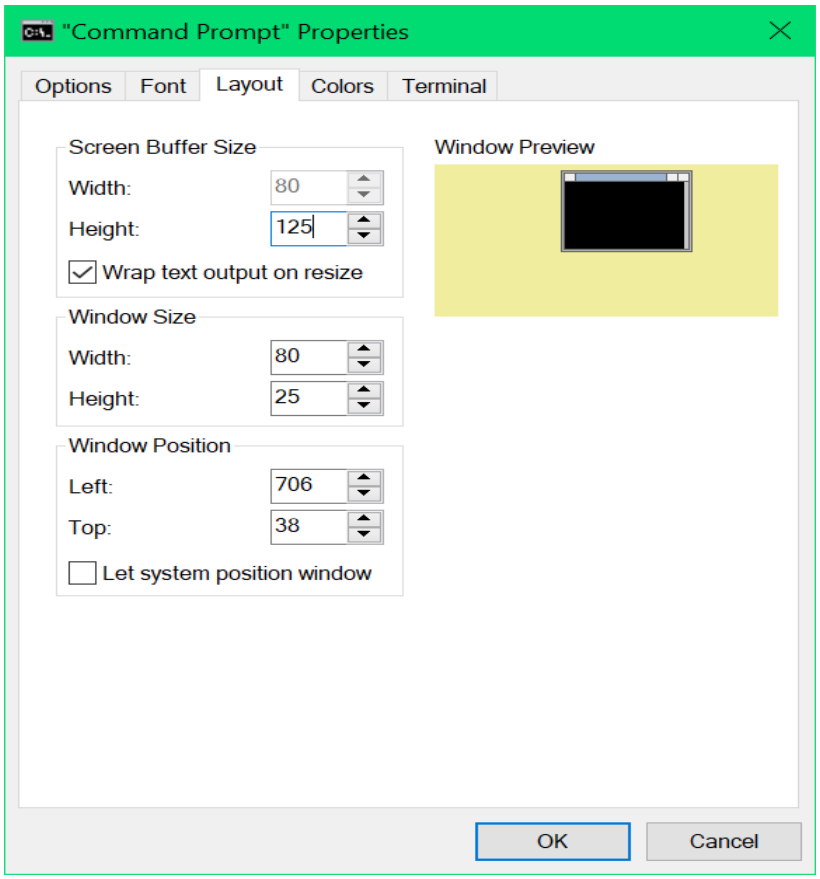
Open a terminal in Linux/MacOs or Command Prompt in Windows; copy the files **rppy.py** and **helprppy.py** to your working directory. Launch with Python, there's no special installing needed:

```
c:\main\Python>python rppy.py
Welcome to Reverse Polish Python - RPPy
  Version 24.11 (yy.mm)
  Type "intro .(Enter)" for introduction, "help .(Enter)" for help,
  "license .(Enter)" for license
  Press Ctrl-Q at line input or "quit .(Enter)" to exit RPPy

 Data stack empty
ex>
```

RPPy was tested under Windows 10 and Python 3.12; it should work at least under Python 3.7 and higher. Further informations in this manual regarding file path and on-screen editing functions are all Windows-specific.

As some console output concerning lists of implemented words is more than a classic 25-line console window height, declaring at least a 125 line output buffer is highly recommended.



RPPy Components

RPPy is implemented on a *virtual machine* **VM** consisting of:
- a **CPU** which executes RPPy's instruction set: the *kernel primitives written in Python*
- an Instruction Pointer **IP**
- a **Data Stack** for passing data between the words that make up a RPPy program
- a **Return Stack** holding continuations (return addresses) of called words
- an **User Stack** for holding intermediary data if desired
- a set of four so-called *registers*:
    - **ZF**: the ZeroFlag register holding comparison results to be used in branching
    - **I, J, K**: the index registers used in sequence treatment and looping

The data stack, user stack and registers are accessible in read/write mode by the user; the return stack is accessible only in read mode; the CPU and IP are not user accessible.

A RPPy *program* consists of a series of *Execution Tokens XT* (names of corresponding kernel primitives) stored in the *Execution List* by compiling user input in the Read-Eval-Print-Loop; the format of the Execution List is *Token Threaded Code* (TTC).
The CPU fetches (loads) XTs from the Execution List accordingly to the Instruction Pointer IP which holds the index of the current XT to be executed. Advancing the IP is done sequentially, except branching.

Attached to the VM are three dictionaries:
- the *kernel primitives dictionary* with key:value pairs of the form:
    - key: RPPy instruction name
    - value: execution token XT (name of attached Python function) & stack effect

      swap: k_swap(), (x y -- y x) exchange TOS with NOS
      +: k_plus(), (x y -- x+y) add TOS to NOS
      count: k_count(), (list value -- n) count instances of value in list

- the *high level words (definitions) dictionary* with key:value pairs of the form:
    - key: definition name
    - value: index in the ExecList where the definition is compiled & attached docstring (documentation string if any)

```
meanval: 2, ( x y -- (x+y)/2 ) compute mean value
2 meanval:          # start of definition in ExecList
3 +
4 lit  2
5 /
6 ;                 # end of definition
```

- the *compiler words dictionary* with key:value pairs of the form:
    - key: name of RPPy word executed at compile phase
    - value: Python function to compile RPPy word & attached docstring

      if: c_compif(), (--) compile if: continue execution if ZF == 1, else branch to "then"

At RPPy starting, the high level definitions dictionary is *always empty*; load your set of saved definitions in previous work sessions or start afresh with a new set, to be saved at session end and reloaded for completion after.

The other two dictionaries, kernel and compile, could be modified only by changing the source rppy.py in further versions.

Display the content of the dictionaries by (see “Using RPPy"):

- **pkernall** or **pkern** for the kernel primitives
- **pdefall** or **pdef** for the high level words
- **pcomp** for the compiling words

You can also look at the *glossary* at manual’s end.

Using RPPy

At program start, RPPy is in *execution mode*, signalled by the prompt "ex> ", awaiting user input; as a following convention, all console I/O is shown in pairs of fenced markers "---" with user input in **bold**; comments may appear after the hash "#"; kernel (built-in) words in text are displayed in **gray bold**.

```
---
 Data stack empty       # there's no data yet in the stack
ex>                     # RPPy prompts for user input
---
```

Now try executing some calculations:

```
---
ex> 10 20 30 .          # put three values on the stack, separated by space(s)
 Data stack items: 3    # ALWAYS terminate input  by dot "." followed by Enter
[10, 20, 30]            # the stack has now three values, with the last one on
ex>                     # top of the stack, shown in the rightmost position
---
```

Add the top two numbers:

```
---
ex> + .
 Data stack items: 2
[10, 50]                # result is on top of stack, replacing arguments 20,30
ex>
---
```

Now divide top of stack (TOS) by next of stack (NOS), aka 50/10

```
---
ex> / .
 Data stack items: 1
[0.2]                   # ?!? that's not what you expected...
ex>
---
```

How does "/" work? Ask by using **pdef**, which means "print definition":

```
---
ex> '/ pdef .           # prefix a string without spaces with single quote "'"
Kernel primitive: / ; XT: k_slash() ; action: ( x y -- x/y ) divide NOS
by TOS - floating point div
 Data stack items: 1
[0.2]
ex>
---
```

Ignore for the moment what it says about kernel primitive and XT, and look at action; the parentheses include always the stack effect of given word: "(*what's on the stack before* -- *what's on the stack after*)", so if you have two numbers x and y before executing division, the "/" operator works by dividing x by y, aka NOS by TOS, not what we wanted: 50/10

Now comes what is known as the biggest hurdle in learning RPPy: *stack shuffling* - a set of words to manipulate stack content in order to achieve desired order of operands needed by each word at execution.

Let's start again:

```
---
ex> 10 20 30 + .       # add the two top numbers
 Data stack items: 3
[0.2, 10, 50]
ex> swap .             # swap does: ( x y -- y x)
 Data stack items: 3
[0.2, 50, 10]          # now order of operands is ok for division
ex> / .                # division
 Data stack items: 2
[0.2, 5.0]             # desired result achieved
ex> swap drop .        # but the wrong result "0.2" should be discarded
                       # so use drop ( x -- ) which discards TOS
 Data stack items: 1
[5.0]                  # finally print result if you wish
ex> print .            # print needs an item to display: ( item -- )
5.0
 Data stack empty      # print consumed TOS, stack is now empty
ex>
---
```

Now suppose you want to keep the above inputted words for further use (even if it doesn't make great sense, only as an example...) ; you create a definition by giving a name with colon ":" as suffix :

```
---
ex> mydef: "" ( n1 n2 n3 -- (n2+n3)/n1 )""    # stack effect
co> + swap / print ;  # words terminated by a semicolon
co> .                 # the dot signals end of definition
 Data stack empty
ex>
---
```

Always put the definition name as the *first item in the input line*, followed (not compulsory, but highly recommended) by the *stack effect*, started and ended by the double double quote '""'. As RPPy has no notion of source code, this stack effect description is the *only documentary part which remains attached to a definition*. Comments, as in Python, could be written by a hash '# ' followed by any text, but they are not taken into account in the dictionary of stored definitions.

If the first item inputted is a definition name, RPPy changes to *compile mode*, signalled by "co>" in the next lines . Now you can input the body of the definition, consisting of *already predefined words* and literals, there's no notion of forward references "use it now, define it later".

You create a program in RPPy by defininig more an more complex words based on the previous ones, a process known as "incremental compiling".

The semicolon "**;**" is the equivalent of a return in Python, but compiling continues until dot "**.**"; think of a sequence of RPPy words as an *ordinary text sentence, ending always with "**.**"* Only after receiving the ending dot RPPy changes back to execute mode. So you can *spread a definition in a number of lines*, but, opposed to Python, there's no indentation. The notion

of code block for each indentation level in Python doesn't exist: you simply give a name to whatever group of words seems appropiate.
OK, after all that , let's see if our definition works:

```
ex> 15 75 25 mydef .
6.666666666666667
 Data stack empty
ex>
```

Wonderful, but what if we forget an argument?

```
ex> 15 75 mydef .
"swap" aborted:  Data Stack Underflow
 Index:  6
 In definition: "mydef" at index: 4
 Return stack items:1
1 return from: mydef
 Data stack empty
ex>
```

Not very encouraging...let's see what happened:
- Data Stack Underflow: you tried to use more arguments than present on the stack
- Index: all RPPy words are encoded in a list called "Execution List", which can be displayed giving starting index followed by **pexlst** , "print execution list". As mydef is coded starting at index 4 and the error-emitting word, swap, is at index 6, you do:

```
ex> 4 pexlst .
   4 mydef:             # start of mydef
   5 +                  # each word comprising mydef is coded as an
   6 swap               # entry in the execution list
   7 /
   8 print
   9 ;                  # end of mydef
  10 lit  15            # what follows after is all what you inputted
  11 lit  75            # to execute mydef
  12 lit  25            # the input "15 75 25 mydef" is coded as three
  13 call mydef         # literals followed by a call to mydef
  14 lit  15            # next, you give only two literals as arguments
  15 lit  75
  16 call mydef         # with only 2 arguments, mydef executed "+" ,
  17 lit  4             # which leaved 90 on the stack, now "swap" tried
  18 pexlst             # to swap 90 with a non-existing value: aborted
 Data stack items: 2
[90, None]
ex>
```

What if we call mydef with a string instead a number?

```
ex> 15 75 " abc" mydef .
```

```
"+" aborted: unsupported operand type(s) for +: 'int' and 'str'
 Aborted at Execution List Index:  5
 In definition: "mydef" at index: 4
  TOS:  <class 'str'> abc
  NOS:  <class 'int'> 74
 Return stack items:1
1 return from: mydef
 Data stack items: 5
[90, None, 15, 75, 'abc']
---
```

This time it's the sum of 75 and the string abc which caused the abort, as an integer cannot be added to a string; in this case, RPPy displayed TOS and NOS, showing the type of operands ("str"-string, "int"-integer)

An *abort* is the equivalent of Python's error messages, but instead of ending the script with a traceback, in RPPy you simply continue working, testing definitions until ok, without the need to edit/reinterpret source code.

Anytime you can save your set of definitions and reload them later for further completion. See "Editing, Saving and Loading Definitions".

Finally, if you want to see later your definition, use **pdef** ;
if a string is spacefree, instead of "(space) mydef" you can use the single quote as prefix, WITHOUT space : 'mydef (see strings in "Basic Datatypes - Strings")

```
---
ex> 'mydef pdef .
  mydef       at index:  4 "" ( n1 n2 n3 -- (n2+n3)/n1 ) ""
  mydef: + swap / print ;

  No duplicate (older definitions) present
 Data stack items: 5
[90, None, 15, 74, 'abc']
ex>
---
```

If you want to see the *dictionary of built-in words*, the so-called *kernel primitives*, enter:

```
---
ex> pkernall .          # pkernall stands for "print all kernel definitions"
  ###1      ===Data Stack words===
  dup         ( n -- n n ) duplicate TOS
  2dup        ( x y -- x y x y ) duplicate NOS and TOS
  drop        ( n -- ) drop TOS
  2drop       ( x y -- ) drop TOS and NOS
  swap        ( x y -- y x ) exchange TOS with NOS
  over        ( x y -- x y x ) copy NOS over TOS
  ...
  ...
---
                          # press Enter to continue displaying, any other key stops
```

```
---
 ###3      ===Math words===
 +          ( x y -- x+y ) add TOS to NOS
 -          ( x y -- x-y ) subtract TOS from NOS
 *          ( x y -- x*y ) multiply TOS by NOS
 /          ( x y -- x/y ) divide NOS by TOS - floating point div
 //         ( x y -- x//y ) divide NOS by TOS - floored (integer) div
 %          ( x y -- rem ) remainder of x/y division
 divmod     ( x y -- quot rem ) quotient & remainder of x/y division
 **         ( x y -- x**y ) NOS at power of TOS
 ++         ( n -- n+1 ) increment TOS by 1
 --         ( n -- n-1 ) decrement TOS by 1
 ...
 ...
---
```

If you want to see a single group of words per category, use **pkern**.

The notation used for the stack effect is as follows:
- **n, x, y, z** :
  - concerning stack manipulation: any type of data
  - concerning math, assignment, bitwise ops : numbers
- **str** : strings
- **seq** : sequence, as strings, lists, etc.
- **idx** : index of a word in the execution list (function pointer)
- **item** : one element in a sequence or anything depending on the using word
- **ZF** : Zero Flag, positioned by comparisons
- **I, J, K** : index registers used to acces/slice sequences and in counted loops

Clear the data stack with **cleards** , there's no need to accumulate a bunch of already used data here; preserve only what you need for further testing:

```
---
ex> cleards . # clears the data stack
 Data stack empty
ex>
---
```

And the famous Hello World! program?

```
---
ex> hello: "" ( -- ) ""       # no data before/after executing "hello"
co> " Hello World!" print ; .
 Data stack empty
ex> hello .
Hello World!
 Data stack empty
ex>
---
```

Displaying all your definitions with **pdefall**:

```
---
ex> pdefall .
 mydef      at index:  4 "" ( n1 n2 n3 -- (n2+n3)/n1 ) ""
```

```
 hello       at index:  10 "" ( -- )  ""

High Level Dictionary Definitions:  2
Data stack empty
ex>
---
```

Now it's time to nail down some basic concepts:

1. a word is *ANY sequence of characters separated by whitespace* (space, tabulation or newline) , there are no "identifiers" as in Python, which must start with a letter or underscore and contain only letters, underscore or the digits 0..9
Valid words are 2dup, sum+ , s[i:j], even n! or $#.k00->/ if that means anything to you...

2. every word has an *ACTION* associated to it, even if at a first glance it doesn't seem so: ok, swap interchages TOS with NOS, but what does 10 or 30 do? Those are *LITERALS*, whose action is always the same: *pushing themselves on the stack*. Numbers, strings, lists, dictionaries, etc. all are literals.

3. knowing the action of a given word, to ensure correct execution of said action means *preparing the stack in advance* BEFORE invoking the word - remember this as the basic rule of concatenative programming!

4. the *evaluation rule* for a concatenative language is very simple: *scan input left to right, push literals on the stack, call words*.
  Each word is either a "*primitive*", coded in Python, or a "*high level*" composed of primitives and other pre-defined high level words.
  There are neither expressions nor statements in RPPy: only a stream of literals and words.

To summarise the rules of RPPy:

1. test a sequence of words in *execute mode*, always looking at the data stack:
- work step by step, input only what's needed for the following word; don't chain more words on a single input line until all except the last one are already tested
- after the sequence of words is tested, *compile it as a new definition*

2. any sequence of literals and words can be *memorised as a definition*:
- always start with the definition's name suffixed with ":"
- the definition name cannot start with:
  - single quote **'** : the prefix for strings without any space included
  - asterisk ***** : the prefix for a word's index in the execution list
- the definition name must be the first item in the input line
- the body of the definition can be introduced on multiple lines, until the ending dot “.”
- attach a stack effect description "" (stack before -- stack after)""

as it'll be the only way after to know what that word does
- exit with dot "." from compile mode
NB. in execution mode, but only here, you can terminate input with *two consecutive* Enter keys; in compile mode the dot is compulsory.

3. a definition can be *redefined with the same name*, but attention:
- new words created from here onwards using the redefined word will, as expected, use the new variant
- older words including this definition will preserve her *old behavior*, there's no automatic replacing of the old ones with the new one; see "Editing, Saving and Loading Definitions"
- as a matter of fact, redefine at will your definition until it's thoroughly tested; only then use it in other definitions. There's no problem if later on you wish to modify this definition, but at the cost of manually replacing the old variant.

4. save your work often, by using **Ctrl-S** or **save**, as there may be situations when RPPy crashes unexpectedly (hopefully not too often…) or simply exits with a Python traceback because of a (not yet) catched error in a kernel primitive; unfortunately at this instant all definitions created in the current session are lost; see "Editing, Saving and Loading Definitions"

5. corollary:
- keep definitions *short*, a couple of lines maximum, operating on a few values once; each time you feel that it gets too complicated, break the definition in more subwords *factoring* as much as possible
- each definition should do only *one thing*
- try to give *meaningful names* to definitions; as there's no source code in RPPy, a good choice of names helps further maintenance
- *minimise stack shuffling*, the stack output of one word should match the stack input of the next word concerning order of items
- *comment the stack effect*, as it's the only comment which remains attached to the definition; you can even add other meaningful information here (only it has to be done on a single input line)

NB1. Read first "Data stack management" and "Error handling" until you gain some experience with RPPy.

NB2. See "A Few Examples – Square Root With Newton's Method" as how to define a small app from simple to more complex.

Data Stack Management

The biggest difficulty in learning RPPy is the proper use of the data stack. As every word uses it to get arguments and to leave results here for the next word, it's crucial to have:

a) correct *number* of arguments:
- too many: stack overflow as unwanted data accumulates in time or wrong input arguments picked
- too few: stack underflow or wrong input arguments picked

b) correct *type* of arguments:
- execution errors if, as example, an integer is given as a filename or a string is added to a number

c) correct *order* of arguments:
- difficult to debug, as, contrary to a) and b), there may be no execution error but instead an erroneous data output

By the way, all words use *positional arguments*, there are no keyword arguments as in Python, so the order of them on the stack is essential.

In order to achieve all this, a number of stack management (or stack "shuffling") words are defined in RPPy; list them with **pkern** option 1:

dup	(n -- n n) duplicate TOS
2dup	(x y -- x y x y) duplicate NOS and TOS
drop	(n --) drop TOS
2drop	(x y --) drop TOS and NOS
swap	(x y -- y x) exchange TOS with NOS
over	(x y -- x y x) copy NOS over TOS
nip	(x y -- y) drop NOS
tuck	(x y -- y x y) insert TOS before NOS
rot	(x y z -- y z x) rotate rightwise top three items
-rot	(x y z -- z x y) rotate leftwise top three items
pick	(xn xn-1 xn-2 ... x0 n -- xn xn-1 xn-2 ... x0 xn) replace TOS with a copy of the n-th item
cleards	(... --) empty data stack

Here TOS means Top Of Stack and NOS is Next Of Stack.

Usually you'll be using dup/drop to get desired number of arguments and swap/over/rot to get desired order, but that's only a very, very simplistic rule of thumb...

See some examples by using RPPy as a calculator:

```
ex> # calculate 7*3**2-2*3 given 7 3 2 on the stack
ex> 7 3 2 .          # initial values
 Data stack items: 3
[7, 3, 2]
ex> over .
 Data stack items: 4
[7, 3, 2, 3]
```

```
ex> * .
 Data stack items: 3
[7, 3, 6]
ex> swap .
 Data stack items: 3
[7, 6, 3]
ex> dup .
 Data stack items: 4
[7, 6, 3, 3]
ex> * .
 Data stack items: 3
[7, 6, 9]
ex> rot .
 Data stack items: 3
[6, 9, 7]
ex> * .
 Data stack items: 2
[6, 63]
ex> swap .
 Data stack items: 2
[63, 6]
ex> - .
 Data stack items: 1
[57]                # final result
ex> drop .          # discard it
 Data stack empty
ex>
```

Of course you could have done it in a single input line after 1 year of experience :):)

```
ex> 7 3 2 over * swap dup * rot * swap - .
 Data stack items: 1
[57]
ex>
```

```
ex> # calculate the sum of squares 10**2 + 20**2 + 30**2
ex> 10 20 30 .          # initial values
 Data stack items: 3
[10, 20, 30]
ex> dup .
 Data stack items: 4
[10, 20, 30, 30]
ex> * .
 Data stack items: 3
[10, 20, 900]
ex> swap .
 Data stack items: 3
[10, 900, 20]
ex> dup .
```

```
 Data stack items: 4
[10, 900, 20, 20]
ex> * .
 Data stack items: 3
[10, 900, 400]
ex> + .
 Data stack items: 2
[10, 1300]
ex> swap .
 Data stack items: 2
[1300, 10]
ex> dup .
 Data stack items: 3
[1300, 10, 10]
ex> * .
 Data stack items: 2
[1300, 100]
ex> + .
 Data stack items: 1
[1400]                # result
ex>
---
```

A good help if it gets too complicated is the *user stack* to save intermediary data: as example define a word **2swap**

```
---
ex> 2swap: "" ( a b c d -- c d a b)""
co> rot             #  ( a b c d -- a c d b )
co> uspush          #  ( a c d b -- a c d ) push TOS to user stack
co> rot             #  ( a c d -- c d a )
co> uspop           #  ( c d a -- c d a b ) pop user stack to TOS
co> ; .
 Data stack empty
ex> 11 22 33 44 .
 Data stack items: 4
[11, 22, 33, 44]
ex> 2swap .
 Data stack items: 4
[33, 44, 11, 22]
ex> '2swap pdef .
  2swap       at index:  2 "" ( a b c d -- c d a b) ""
  2swap: rot uspush rot uspop ;

  No duplicate (older definitions) present
 Data stack items: 4
[33, 44, 11, 22]
ex>
---
```

Now, with the help of **2swap** define **2over**:

```
---
ex> 2over: "" ( a b c d -- a b c d a b )""
co> uspush uspush       # ( a b c d -- a b )
co> 2dup                # ( a b -- a b a b )
co> uspop uspop         # ( a b a b -- a b a b c d )
co> 2swap               # ( a b a b c d -- a b c d a b )
co> ; .
ex> cleards .
 Data stack empty
ex> 1 2 3 4 .
 Data stack items: 4
[1, 2, 3, 4]
ex> 2over .
 Data stack items: 6
[1, 2, 3, 4, 1, 2]
ex>
---
```

And, if you feel in good mood, define **3dup** and **4dup**:

```
---
ex> 3dup: "" ( a b c -- a b c a b c )""
co> dup                 # ( a b c -- a b c c )
co> 2over               # ( a b c c -- a b c c a b )
co> rot                 # ( a b c c a b -- a b c a b c )
co> ; .
 Data stack empty
ex> 1 2 3 .
 Data stack items: 3
[1, 2, 3]
ex> 3dup .
 Data stack items: 6
[1, 2, 3, 1, 2, 3]
ex> 4dup: "" ( a b c d -- a b c d a b c d )""
co> 2over               #  ( a b c d -- a b c d a b )
co> 2over               #  ( a b c d a b -- a b c d a b c d )
co> ; .
ex> 1 2 3 4 .
 Data stack items: 4
[1, 2, 3, 4]
ex> 4dup .
 Data stack items: 8
[1, 2, 3, 4, 1, 2, 3, 4]
ex>
---
```

Define a word to compute: **$z=ax^2+bx+c$**

```
---
ex> z: "" ( a b c x -- ax**2+bx+c )""
co> dup          # ( a b c x -- a b c x x )
co> 2 **         # ( a b c x x -- a b c x x**2 )
```

```
co> uspush          # ( a b c x x**2 -- a b c x )
co> rot             # ( a b c x -- a c x b )
co> *               # ( a c x b -- a c bx )
co> +               # ( a c bx -- a bx+c )
co> swap            # ( a bx+c -- bx+c a )
co> uspop           # ( bx+c a -- bx+c a x**2 )
co> *               # ( bx+c a x**2 -- bx+c ax**2 )
co> +               # ( bx+c ax**2 -- ax**2+bx+c )
co> ; .             # end word
 Data stack empty
ex> # testing "z"
ex> 2 3 4 1 z .
 Data stack items: 1
[9]
ex> 1 1 1 10 z .
 Data stack items: 2
[9, 111]
ex> 'z pdef .
  z          at index:  45 "" ( a b c x -- ax**2+bx+c ) ""
  z: dup 2 ** uspush rot * + swap uspop * + ;
---
```

For those who don't want headaches with stackrobatics, use variables like good ol' Python (see "Constants, Variables and Assignment Operations")

```
---
ex> a: 0 ;                  # define variables a,b,c,x
co> b: 0 ;
co> c: 0 ;
co> x: 0 ;
co> z: a x 2 ** * b x * + c + ; .     # "z" looks more "formal" with variables...
 Data stack empty
ex> 2 *a = 3 *b = 4 *c = 1 *x = . # assign values to variables prefixed with "*"
 Data stack empty
ex> a b c x .               # list variables (only to see them, not needed on the stack
 Data stack items: 4        # by "z")
[2, 3, 4, 1]
ex> z .                     # get result
 Data stack items: 5
[2, 3, 4, 1, 9]
ex>
---
```

Beware that **2dup** is not the equivalent of **dup dup**:

```
---
ex> 99 88 2dup .
 Data stack items: 4
[99, 88, 99, 88]
ex> cleards .
 Data stack empty
ex> 99 88 dup dup .
 Data stack items: 4
```

```
[99, 88, 88, 88]
ex>
---
```

A word about **pick**: even if it's part of RPPy, use it sparely: do not treat the data stack as an array (internally it's a list, but whatever) where you can pick at will at any depth; a stack does it's job by pushing and popping data, not by indexing. By the way, "0 pick" is **dup** and "1 pick" is **over** ...

```
---
ex> 11 22 33 .
 Data stack items: 3
[11, 22, 33]
ex> 0 pick .
 Data stack items: 4
[11, 22, 33, 33]
ex> drop .
 Data stack items: 3
[11, 22, 33]
ex> 1 pick .
 Data stack items: 4
[11, 22, 33, 22]
ex>
---
```

A good way to minimise stack shuffling is combining a group of short definitions instead of a single large one; also think of arranging the output of a word so as the next word in chain isn't forced to rearrange it's received input to work correctly.

Constants, Variables, Assignment Operations

There are no constants in RPPy (Python has a few ones: True, False, None, NotImplemented ...) but defining them is easy:

```
ex> PI: 3.14159 ; .
 Data stack empty
ex> PI .
 Data stack items: 1
[3.14159]
ex>
```

Concerning True, False and None: all values which are not zero are True. True, False and None are reflected in RPPy with the predefined ZeroFlag ZF, set to 1 for True and to 0 for False & None; see "Control Flow".

There are no variables in RPPy (?!?) ; ok, there are no *explicit* variables, any definition whose *first item is a literal* is a variable; let's see an example:
In Python:

```
>>> a = 10
>>> b = 20
>>> c = a + b
>>> c
30
>>>
```

And now in RPPy:

```
ex> a: 10 ;
co> b: 20 ;
co> c: a b + ; .
 Data stack empty
ex> c .
 Data stack items: 1
[30]
ex>
```

Assigning a new value to a variable:

```
ex> 15 *a = .          # assign 15 to indexed variable a
 Data stack items: 1   # always prefix with "*" variable's name
[30]                   # to obtain the index (pointer) to that variable
ex> 25 *b = .          # idem, assign 25 to b
 Data stack items: 1
[30]
ex> c .                # executing c shows the new value 40
 Data stack items: 2
```

```
[30, 40]
ex>
---
```
The asterisk "*" works by leaving on the stack the *index*, or pointer if you prefer, where the variable is defined in the execution list. Now, the equal "=" sign , meaning assigning as in Python, tests if the next item in the definition is a literal; if ok, the new value is stored in the definition, so next time it's executed the new value is pushed on the stack. Forgetting to prefix a variable aborts the assignment, with all sort of strange output depending on what "=" finds at the false index position in the execution list…

By the way, as there is no (not yet...) notion of namespace in RPPy, all variables are, being nothing more than a definition, global, like all definitions are.

There's much less need to work with variables in RPPy compared to Python; pass arguments for words on the stack, not in variables.

If you want to avoid using variables as locals in words, RPPy offers an auxiliary data stack, the *user stack*, accessed by:

```
  uspush      ( n -- ) data stack  | ( -- n ) user stack ; push TOS to user stack
  uspop       ( -- n ) data stack  | ( n -- ) user stack ; pop TOS of user stack
                                       & push to data stack
  clearus     ( ... -- ) empty user stack
  pus         ( -- ) print user stack, not the fluid produced by inflammatory
                     infections...
```

```
---
ex> 88 99 .
 Data stack items: 2
[88, 99]
ex> uspush .
 Data stack items: 1
[88]
ex> uspush .
 Data stack empty
ex> pus .
 User stack items: 2
[99, 88]
 Data stack empty
ex> uspop pus .
 User stack items: 1
[99]
 Data stack items: 1
[88]
ex> uspop pus .
 User stack empty
 Data stack items: 2
[88, 99]
---
```

Following assignment operations are available in RPPy (see **pkern**):

```
###7      ===Assignment words===
=          ( n idx -- ) store n to var[idx]: variable with index idx
+=         ( n idx -- ) var[idx] += n
-=         ( n idx -- ) var[idx] -= n
*=         ( n idx -- ) var[idx] *= n
/=         ( n idx -- ) var[idx] /= n
//=        ( n idx -- ) var[idx] //= n
%=         ( n idx -- ) var[idx] %= n
**=        ( n idx -- ) var[idx] **= n
```

Always push the desired value on the stack, followed by the indexed variable.

```
ex> a b .
 Data stack items: 2
[15, 25]                    # old values of a,b
ex> 3 *a /=   2 *b *= .     # divide a by 3, multiply b by 2
 Data stack items: 2
[15, 25]
ex> a b .
 Data stack items: 4
[15, 25, 5.0, 50]           # new values of a,b
ex>
```

NB. You *cannot index a kernel primitive*, as it's not defined in the Execution List, so it has no index. But you can define a word containing respective primitive and index it:

```
ex> *+ .                              # "+" is in the kernel dictionary, not in the high
Compile error: word "+" not found     # level definitions dictionary, therefore * doesn't
ex> plus: + ; .                       # find it
 Data stack empty
ex> *plus .                           # define a word containing desired primitive
 Data stack items: 1                  # and show respective index
[276]
ex> cleards .
 Data stack empty
ex> 10 20 + .                         # adding top two values with kernel primitive
 Data stack items: 1
[30]
ex> 10 20 *plus execidx .             # adding top two values by using execidx and
 Data stack items: 2                  # index of high level word containing "+"
[30, 30]
ex>
```

Basic Datatypes - Numbers

Three types available:
- integers
- floating point
- complex

```
ex> 22 33 +
ex>
 Data stack items: 1
[55]
ex> type .                    # use type to get type of data
 Data stack items: 1
[<class 'int'>]               # int for integer type
ex> 44 11 / .
 Data stack items: 2
[<class 'int'>, 4.0]          # division leaves always a floating point number
ex> type .
 Data stack items: 2
[<class 'int'>, <class 'float'>] # float type
ex> 20 6 // .                 # floored division, leaves integer part
 Data stack items: 3
[<class 'int'>, <class 'float'>, 3]
ex> (2+6j) (3+4j) + .         # complex numbers
 Data stack items: 4
[<class 'int'>, <class 'float'>, 3, (5+10j)]
ex> type .
 Data stack items: 4
[<class 'int'>, <class 'float'>, 3, <class 'complex'>]
ex>
```

Look with **pkern** to see the available math operations:

```
  ###3      ===Math words===
+       ( x y -- x+y ) add TOS to NOS
-       ( x y -- x-y ) subtract TOS from NOS
*       ( x y -- x*y ) multiply TOS by NOS
/       ( x y -- x/y ) divide NOS by TOS - floating point div
//      ( x y -- x//y ) divide NOS by TOS - floored (integer) div
%       ( x y -- rem ) remainder of x/y division
divmod ( x y -- quot rem ) quotient & remainder of x/y division
**      ( x y -- x**y ) NOS at power of TOS
++      ( n -- n+1 ) increment TOS by 1
--      ( n -- n-1 ) decrement TOS by 1
neg     ( n -- neg(n) ) negate n
abs     ( x -- abs(x) ) absolute value of x
round   ( n i -- round(n))  n rounded to i digits
...
...
```

Some math operations examples:

```
ex>  4 3 / .                  # division
 Data stack items: 1
[1.3333333333333333]
ex> 4 3 // .                  # floored division
 Data stack items: 2
[1.3333333333333333, 1]
ex> 4 3 % .                   # remainder of division
 Data stack items: 3
[1.3333333333333333, 1, 1]
ex> 4 3 divmod .              # quotient and remainder
 Data stack items: 5
[1.3333333333333333, 1, 1, 1, 1]
ex> 2drop 2drop .
 Data stack items: 1
[1.3333333333333333]
ex> 2 round .                 # round float to 2 digits
 Data stack items: 1
[1.33]
ex> neg .                     # negate value
 Data stack items: 1
[-1.33]
ex> abs .                     # absolute value
 Data stack items: 1
[1.33]
ex> [1,2,3,4,5] min .         # minimum of values in a sequence
 Data stack items: 2
[1.33, 1]
ex> [1,2,3,4,5] max .         # maximum of values in a sequence
 Data stack items: 3
[1.33, 1, 5]
ex> cleards [1,2,3,4,5] sum . # sum of values in a sequence
 Data stack items: 1
[15]
ex> 5 4 3 2 1    5 val+ .     # sum of top 5 values
 Data stack items: 2
[15, 15]
ex> 5 4 3 2 1    5 val* .     # product of top 5 values
 Data stack items: 3
[15, 15, 120]
ex> " abcd " " efgh " " ijkl " 3 str+ .    # concatenate top 3 strings with spaces
 Data stack items: 4
[15, 15, 120, 'abcd efgh ijkl ']
ex> 'abcd 'efgh 'ijkl 3 str+ .             # idem without spaces
 Data stack items: 5
[15, 15, 120, 'abcd efgh ijkl ', 'abcdefghijkl']
ex> cleards .
 Data stack empty
ex> [1,2,3] ["abcd","efgh","ijkl"] 2 lst+ . # concatenate top 2 lists
 Data stack items: 3
```

```
[[1, 2, 3], ['abcd', 'efgh', 'ijkl'], [1, 2, 3, 'abcd', 'efgh', 'ijkl']]
ex>
---
```

Converting to different number types is available with:

int	(str/n -- n) convert str or number n to integer n
float	(str/int -- n) convert string or integer to floating point number
complex	(str -- n) convert str to complex number n
str	(n -- str) return a string interpretation of n

```
---
ex> " 12"  int .                  # str to int
 Data stack items: 1
[12]
ex> 12.66 int .                   # float to int
 Data stack items: 2
[12, 12]
ex> 12 float .                    # int to float
 Data stack items: 3
[12, 12, 12.0]
ex> cleards .
 Data stack empty
ex> " (12+24j)" complex .         # str to complex
 Data stack items: 1
[(12+24j)]
ex> " 12" complex .               # str to complex – real part only
 Data stack items: 2
[(12+24j), (12+0j)]
ex> " 24j" complex .              # str to complex – imaginary part only
 Data stack items: 3
[(12+24j), (12+0j), 24j]
ex> " 12+24j" complex .           # str to complex, parentheses not necessary
 Data stack items: 4
[(12+24j), (12+0j), 24j, (12+24j)]
ex> cleards .
 Data stack empty
ex> 12.66 str .                   # number to str
 Data stack items: 1
['12.66']
ex> dup type .
 Data stack items: 2
['12.66', <class 'str'>]
ex> drop .
 Data stack items: 1
['12.66']
ex>
---
```

See also "Miscellaneous words" for other types of number conversions.

Basic Datatypes - Strings

Strings can be defined in RPPy as follows:
- if the string doesn't have any space included, *prefix* it with single quote

---

```
ex> 'Hello,World! .
 Data stack items: 1
['Hello,World!']
ex>
```

- if the string has spaces included, use the *word* double quote " as start of string: the space after " , as you already know, is the word's separator and is obviously not part of the string. The first next double quote encountered ends the string: that means that you cannot include double quotes as part of the string, only single quotes

---

```
ex> " Hello World!   " .
 Data stack items: 2
['Hello,World', 'Hello World!   ']
ex> "     Hello World!" .
 Data stack items: 3
['Hello,World', 'Hello World!   ', '    Hello World!']
ex> " Hello 'World!'" .
 Data stack items: 4
['Hello,World', 'Hello World!   ', '    Hello World!', "Hello 'World!'"]
ex>
```

- if the string includes a mix of single and double quotes,use the word **->** as starting the string, and the marker (not word!) **<-** as end of string

---

```
ex> cleards .
 Data stack empty
ex> -> all sort of quotes: "doubles 'singles' mixed"   <- .
 Data stack items: 1
['all sort of quotes: "doubles \'singles\' mixed"   ']
ex> print .
all sort of quotes: "doubles 'singles' mixed"
 Data stack empty
ex>
```

NB. the backslash "\" is used to quote a quote (escape it) at displaying the stack; print interprets the quoted quote correctly

- if the string starts with the word """ double double quote, and ends with the """ marker, it is a *docstring* attached to the current definition. Using it outside a definition ignores all the rest of the input line

---

```
ex> spam: " that's the spam definition" print
```

```
co> " nothing to do"
co> "" (spam before -- spam after)"" ; .
 Data stack empty
ex> spam .
that's the spam definition
 Data stack items: 1
['nothing to do']
ex> 'spam pdef .
  spam       at index:  2 "" (spam before -- spam after) ""
  spam: " that's the spam definition" print " nothing to do" ;

  No duplicate (older definitions) present
 Data stack items: 1
['nothing to do']
ex>
---
```

- if a *multiline string* is needed, start it with the word """ triple double quote and end it with the triple quote marker """ in the last line. All input data after the end marker """ is lost, *put always the end marker on a separate input line*

```
---
ex> cleards .
 Data stack empty
ex> """ Multiline string
""> line 1              # apart "ex>" and "co>" , this is the only case when
""> line 2              # at input the prompt '"">' is displayed, until ending
""> line 3              # the multiline string
""> end """ 111 222 .   # 111 222 and the dot are ignored!
Compile warning: all data after closing triple-quote in current line is ignored!
ex> .                 # so, to end input, press "."
 Data stack items: 1
['Multiline string \nline 1\nline 2\nline 3\nend ']
ex> print .            # the "\n" is the escape character for newline
Multiline string       # print interprets correctly "\n"
line 1
line 2
line 3
end
 Data stack empty
ex>
---
```

Strings, being a sequence of characters, can be indexed using the built-in *index registers* **I, J, K**. An index register is like a variable, but shorter at use:

```
---
ex> 0 i= .          # no need for an "*" prefixing i; no need for separate "="
 Data stack empty
ex> i .             # like a variable, i leaves it's value on the stack
 Data stack items: 1
```

```
[0]
ex>
---
  The first item in a sequence has index 0:
---
ex> " Hello!" .
 Data stack items: 1
['Hello!']
ex> s[i] .                # extract character at index i by "s[i]"
 Data stack items: 2     # which stands for "sequence item at i"
['Hello!', 'H']          # "H" is at index 0
ex> swap .               # put the string "Hello" at TOS
 Data stack items: 2     # as needed by s[i]
['H', 'Hello!']
ex> 4 i= s[i] .          # extract the fifth character at index 4
 Data stack items: 3
['H', 'Hello!', 'o']
ex>
---
  The last item in a sequence has index -1, before last -2, a.s.o.:
---
ex> swap .
 Data stack items: 3
['H', 'o', 'Hello!']
ex> -1 i= s[i] .         # extract "!"
 Data stack items: 4
['H', 'o', 'Hello!', '!']
ex> swap .
 Data stack items: 4
['H', 'o', '!', 'Hello!']
ex> -5 i= s[i] .         # extract "e"
 Data stack items: 5
['H', 'o', '!', 'Hello!', 'e']
ex>
---
  Extracting a substring, called slicing, uses two indexes I and J where
  I = start index, J = end index ( the end index is always excluded):
---
ex> swap .
 Data stack items: 5
['H', 'o', '!', 'e', 'Hello!']
ex> 2 i= 4 j= s[i:j] .  # extract from 2 (included) to 4 (excluded)
 Data stack items: 6
['H', 'o', '!', 'e', 'Hello!', 'll']
ex>
---
  For practical reasons, you can use s[i:] to extract from i to the end,
  and s[:j] to extract from start to j :
---
ex> swap .
 Data stack items: 6
```

```
['H', 'o', '!', 'e', 'll', 'Hello!']
ex> 3 i= s[i:] .        # extract from 3 until end
 Data stack items: 7
['H', 'o', '!', 'e', 'll', 'Hello!', 'lo!']
ex> swap .
 Data stack items: 7
['H', 'o', '!', 'e', 'll', 'lo!', 'Hello!']
ex> 5 j= s[:j] .        # extract from start until 5 (excluded)
 Data stack items: 8
['H', 'o', '!', 'e', 'll', 'lo!', 'Hello!', 'Hello']
ex> swap .
 Data stack items: 8
['H', 'o', '!', 'e', 'll', 'lo!', 'Hello', 'Hello!']
ex> -3 j= s[:j] .       # extract from start until -3 (excluded)
 Data stack items: 9
['H', 'o', '!', 'e', 'll', 'lo!', 'Hello', 'Hello!', 'Hel']
ex>
---
  Using an index out of range aborts s[i] :
---
ex> swap .
 Data stack items: 9
['H', 'o', '!', 'e', 'll', 'lo!', 'Hello', 'Hel', 'Hello!']
ex> 999 i= s[i] .
"s[i]" aborted: string index out of range
 Aborted at Execution List Index:  56
 Aborted in execution string or in definition with multiple returns
  TOS:  <class 'str'> Hello!
  NOS:  <class 'str'> Hel
 Data stack items: 9
['H', 'o', '!', 'e', 'll', 'lo!', 'Hello', 'Hel', 'Hello!']
ex>
---
  However in slicing, an out of range index doesn't leave an error:
---
ex> s[i:] .          # remember, i is 999
 Data stack items: 10
['H', 'o', '!', 'e', 'll', 'lo!', 'Hello', 'Hel', 'Hello!', '']
ex>                 # as there's nothing from 999 onwards, the result
                    # is the empty string '' ...
---

  Strings are immutable in RPPy as in Python, you cannot insert a string in
  another; create a new one by slicing and adding substrings:
---
ex> cleards
ex>
 Data stack empty
ex> " Hello!" .
 Data stack items: 1
['Hello!']
```

```
ex> 5 j= s[:j] .        # extract first part
 Data stack items: 2
['Hello!', 'Hello']
ex> " , World!" + .     # add second part
 Data stack items: 2
['Hello!', 'Hello, World!']
ex> print .
Hello, World!
 Data stack items: 1
['Hello!']
ex>
---
```

A last word about empty strings: you cannot create an empty string using " " or " " ; the first is aborted and the second is a string with lenght 1 containing a space. Use **emptystr** instead

```
---
ex> " " .
Compile error: " without closing marker "
ex> "  " .
 Data stack items: 2
['Hello!', ' ']
ex> emptystr .
 Data stack items: 3
['Hello!', ' ', '']
ex>
---
```

There are many words concerning string manipulations; see **pkern**

Basic Datatypes - Lists

Lists are composed of comma-separated values (items) enclosed by *square brackets*; usually the items are of same type, but they can be mixed of all sort: numbers, strings, other lists, etc.

Creating lists in RPPy can be done as follows:

- if no space follows the separating commas, and no spaces in string items, write it simply as is:

```
ex> [1,2,"abc",5.99,"efg"] .
 Data stack items: 1
[[1, 2, 'abc', 5.99, 'efg']]
ex>
```

- if anywhere there's at least one space present, create a string with all items, *without* the enclosing square brackets and use the word **list**:

```
ex> " 1, 2, 'a b c', 5.99, 'e f g'" list .
 Data stack items: 2
[[1, 2, 'abc', 5.99, 'efg'], [1, 2, 'a b c', 5.99, 'e f g']]
ex>
```

Use single quotes inside the defining string, otherwise the first double quote encountered ends abruptly your string; use **->** and **<-** if you want to mix single with double quotes (see "Basic Datatypes - Strings")

- if you want to create a list from a series of items *already* on the stack, push the starting marker **[]** followed by any items wanted and use **listcre**:

```
ex> '[] 11 22 'abc " e f g h" 33 44 listcre .
 Data stack items: 3
[[1, 2, 'abc', 5.99, 'efg'], [1, 2, 'a b c', 5.99, 'e f g'],
[11, 22, 'abc', 'e f g h', 33, 44]]
ex>
```

Forgetting to push the starting marker **[]** forces listcre to add all data stack content to your list:

```
ex> cleards .
 Data stack empty
ex> 111 222 'abcdefgh " xx yy" 333 listcre .
 Data stack items: 1
[[111, 222, 'abcdefgh', 'xx yy', 333]]
ex>
```

- the content of a given list can be pushed as separate items on the stack using the word **listexp**:

```
---
ex> 111 222 'abcdefgh " xx yy" 333 listcre .
 Data stack items: 1
[[111, 222, 'abcdefgh', 'xx yy', 333]]
ex> listexp .
 Data stack items: 5
[111, 222, 'abcdefgh', 'xx yy', 333]
ex>
---
```

Beware that **listexp** doesn't push a starting marker on the stack...

Lists, being sequences, can be indexed and sliced, like strings, see "Basic Datatypes - Strings" for details on indexing & slicing

```
---
ex> cleards
ex>
 Data stack empty
ex> " 11, 22, 'abc', 33, 'xyz'" list .
 Data stack items: 1
[[11, 22, 'abc', 33, 'xyz']]
ex> 1 i= s[i] .              # extract item 1
 Data stack items: 2
[[11, 22, 'abc', 33, 'xyz'], 22]
ex> swap .
 Data stack items: 2
[22, [11, 22, 'abc', 33, 'xyz']]
ex> 4 j= s[i:j] .            # extract slice from 1 (included) to 4 (excluded)
 Data stack items: 3         # the slice is also a list of 3 elements
[22, [11, 22, 'abc', 33, 'xyz'], [22, 'abc', 33]]
ex>
---
```

Lists, contrary to strings, are *mutable*: you can insert and delete items:

```
---
ex> 2 i= del[i] .            # delete item at index 2 aka 33 from [22,'abc',33]
 Data stack items: 3
[22, [11, 22, 'abc', 33, 'xyz'], [22, 'abc']]
ex> 1 99 insert .            # insert at index 1 the value 99
 Data stack items: 3
[22, [11, 22, 'abc', 33, 'xyz'], [22, 99, 'abc']]
ex> 3 swap insert .          # insert [22,99,'abc'] at index 3
 Data stack items: 2         # now it's a list with a list included
[22, [11, 22, 'abc', [22, 99, 'abc'], 33, 'xyz']]
ex> 3 i= 2 j= s[i][j] .      # get third item from the sublist at index 3 in list
 Data stack items: 3
[22, [11, 22, 'abc', [22, 99, 'abc'], 33, 'xyz'], 'abc']
ex>
---
```

Use the word **len** to get the lenght of a list (or string, dictionary, etc.):

```
ex> drop .                # discard 'abc'
 Data stack items: 2
[22, [11, 22, 'abc', [22, 99, 'abc'], 33, 'xyz']]
ex> len .                 # the lenght is 6, as the sublist counts as a single
 Data stack items: 2      # element, not three
[22, 6]
ex>
```

See **pkern** for the words concerning list manipulation

Basic Datatypes - Dictionaries

Dictionaries are composed of a set of key:value pairs enclosed in *curly braces "{}"* ; accessing a value is done by these keys, not by index as in strings or lists.

Creating a dictionary in RPPy can be done as follows:

- if no space follows the separating commas, and no spaces in string items, write it simply as is:

```
ex> {'jessie':22,'nora':30,'johanna':33} .
 Data stack items: 1
[{'jessie': 22, 'nora': 30, 'johanna': 33}]
ex>
```

- if anywhere there's at least one space present, create a string with all items, *without* the enclosing curly braces and use the word **dict**:

```
ex> " 10:'value ten', 20:'value twenty', 'list':[1, 2, 3]" dict .
 Data stack items: 2
[{'jessie': 22, 'nora': 30, 'johanna': 33}, {10: 'value ten', 20: 'value twenty', 'list': [1, 2, 3]}]
ex>
```

- finally, you can create a dictionary by using *two lists of equal lenght*: the first is a list of keys, the second a list of associated values to that keys; use the word **dictcre**:

```
ex> [1,2,3] ['abcd','efgh','ijkl'] dictcre .
 Data stack items: 1
[{1: 'abcd', 2: 'efgh', 3: 'ijkl'}]
ex>
```

- the reverse of **dictcre** is **dictexp**, which expands a dictionary to the two lists: key list and value list

- to get a value attached to a key, use **get** :

```
ex> dup 2 get .         # duplicate TOS, as get removes it
 Data stack items: 2    # use key "2" to extract "efgh"
[{1: 'abcd', 2: 'efgh', 3: 'ijkl'}, 'efgh']
ex>
```

- to return and remove a value with associated key, use **popkey**; to add a new key:value pair use **setdefault**

```
ex> drop dup 3 popkey .
```

```
 Data stack items: 3
[{1: 'abcd', 2: 'efgh'}, {1: 'abcd', 2: 'efgh'}, 'ijkl']
ex> drop 'one_hundred 100 setdefault .
 Data stack items: 2
[{1: 'abcd', 2: 'efgh', 'one_hundred': 100}, {1: 'abcd', 2: 'efgh', 'one_hundred':
100}]
ex>
```

Important to remember: dup, even it appears to duplicate TOS, for objects like dictionaries, lists, etc. RPPy *duplicates only a reference* to the same object; as a consequence, modifying the dictionary on top modifies also the same dictionary (in fact it's a single object) on next of stack. Use **copy** instead of **dup** if you want preserving the initial data:

```
ex> drop .
 Data stack items: 1    # leave the original dictionary
[{1: 'abcd', 2: 'efgh', 'one_hundred': 100}]
ex> copy .
 Data stack items: 2    # create a copy, not a duplicate with "dup"
[{1: 'abcd', 2: 'efgh', 'one_hundred': 100}, {1: 'abcd', 2: 'efgh', 'one_hundred':
100}]
ex> 'two_hundred 200 setdefault .
 Data stack items: 2    # this time the original data remains unmodified
[{1: 'abcd', 2: 'efgh', 'one_hundred': 100}, {1: 'abcd', 2: 'efgh', 'one_hundred': 100,
'two_hundred': 200}]
ex>
```

- finally, to add a entire sequence to a dictionary, use **update**:

```
ex> agedict: {'jessie':22,'nora':30,'johanna':33} ; .
 Data stack empty
ex> agedict print .
{'jessie': 22, 'nora': 30, 'johanna': 33}
 Data stack empty
ex> agedict " 'hermann':40, 'mike':45, 'josh':50" dict update .
 Data stack items: 1
[{'jessie': 22, 'nora': 30, 'johanna': 33, 'hermann': 40, 'mike': 45, 'josh': 50}]
ex>
```

Basic Datatypes - Tuples, Sets

A tuple is similar to a list, but enclosed in *parentheses*; it may contain data of different types, but contrary to lists, tuples are, like strings, *immutable*.

Creating a tuple in RPPy can be done as follows:
- if no space follows the separating commas, and no spaces in string items, write it simply as is:

```
ex> (1,2,'abc','efg') .
 Data stack items: 1
[(1, 2, 'abc', 'efg')]
ex>
```

- if anywhere there's at least one space present, create a string with all items, *without* the enclosing parentheses and use the word **tuple**:

```
ex> " 1, 2, 3, 'Hello World!', 99" tuple .
 Data stack items: 2
[(1, 2, 'abc', 'efg'), (1, 2, 3, 'Hello World!', 99)]
ex>
```

- finally you can create a tuple by giving as input a list of values and using **tupcre**; expand a tuple to a list of values by **tupexp**:

```
ex> [111,222,'abcd'] tupcre .
 Data stack items: 3
[(1, 2, 'abc', 'efg'), (1, 2, 3, 'Hello World!', 99), (111, 222, 'abcd')]
ex> tupexp .
 Data stack items: 3
[(1, 2, 'abc', 'efg'), (1, 2, 3, 'Hello World!', 99), [111, 222, 'abcd']]
ex>
```

- as with lists, get an item by using **s[i]**; you can also apply slicing, see "Basic Datatypes - Strings"

```
ex> drop 3 i= s[i] .    # drop the list leaved by tupexp
 Data stack items: 3    # get item at index 3
[(1, 2, 'abc', 'efg'), (1, 2, 3, 'Hello World!', 99), 'Hello World!']
ex>
```

- tuples being immutable, you cannot delete or insert an item:

```
ex> drop .
 Data stack items: 2
```

```
[(1, 2, 'abc', 'efg'), (1, 2, 3, 'Hello World!', 99)]
ex> del[i] .                    # try to delete item at index 3 in I
"del[i]" aborted: 'tuple' object doesn't support item deletion
 Aborted at Execution List Index:  25
 Aborted in execution string or in definition with multiple returns
  TOS:  <class 'tuple'> (1, 2, 3, 'Hello World!', 99)
  NOS:  <class 'tuple'> (1, 2, 'abc', 'efg')
 Data stack items: 2
[(1, 2, 'abc', 'efg'), (1, 2, 3, 'Hello World!', 99)]
ex> 4 " Bye!" insert .       # try to insert an item at index 4
"insert" aborted: 'tuple' object has no attribute 'insert'
 Aborted at Execution List Index:  28
 Aborted in execution string or in definition with multiple returns
  TOS:  <class 'str'> Bye!
  NOS:  <class 'int'> 4
 Data stack items: 4
[(1, 2, 'abc', 'efg'), (1, 2, 3, 'Hello World!', 99), 4, 'Bye!']
ex>
---
```

A set is an *unordered collection* of items enclosed in *curly braces*, *without any duplicated elements*, used for membership testing and operations like union, intersection, difference.

Creating a set in RPPy is done, as with tuples, by using **set** for a string with items or simply by enclosing items in curly braces if no spaces present:

```
---
ex> " 'apple', 'banana', 'pear', 'apple', 'apple', 'banana'" set .
 Data stack items: 1
[{'apple', 'banana', 'pear'}]  # all duplicates removed
ex> {1,2,3,1,1,3,4,5,5} .
 Data stack items: 2
[{'apple', 'banana', 'pear'}, {1, 2, 3, 4, 5}]
ex>
---
```

Also, use **setcre** to create a set from a list, and **setexp** to expand the set to a list, as with tuples above.

A string itself can be decomposed to it's characters by using **setstr**:

```
---
ex> " abcd efgh ijkl ab" setstr .
 Data stack items: 3
[{'apple', 'banana', 'pear'}, {1, 2, 3, 4, 5}, {' ', 'c', 'l', 'd', 'j',
'b', 'g', 'f', 'e', 'i', 'h', 'a', 'k'}]
ex> 'abracadabra setstr .
 Data stack items: 4
[{'apple', 'banana', 'pear'}, {1, 2, 3, 4, 5}, {' ', 'c', 'l', 'd', 'j',
'b', 'g', 'f', 'e', 'i', 'h', 'a', 'k'}, {'c', 'r', 'd', 'b', 'a'}]
ex>
---
```

Difference of two sets:

```
---
ex> cleards
ex>
 Data stack empty
ex> 'abcd setstr 'cdef setstr .
 Data stack items: 2
[{'d', 'b', 'c', 'a'}, {'d', 'c', 'f', 'e'}]
ex> difference .
 Data stack items: 1
[{'b', 'a'}]
ex>
---
```

Symmetric difference:

```
---
ex> drop .
 Data stack empty
ex> 'abcd setstr 'cdef setstr .
 Data stack items: 2
[{'d', 'b', 'c', 'a'}, {'d', 'c', 'f', 'e'}]
ex> sdifference .
 Data stack items: 1
[{'b', 'f', 'e', 'a'}]
ex>
---
```

Control Flow

Conditions

As in Python, control flow is based on decisions and decisions are based on comparisons; the first set is identically to Python:

<	(x y -- x y) ZF = 1 if x<y
>	(x y -- x y) ZF = 1 if x>y
<=	(x y -- x y) ZF = 1 if x<=y
>=	(x y -- x y) ZF = 1 if x>=y
==	(x y -- x y) ZF = 1 if x==y
!=	(x y -- x y) ZF = 1 if x!=y

But, opposed to Python, there are no True/False boolean values; instead in RPPy you have the *ZeroFlag register*, **ZF**, which is set to 1 or 0 and tested by branching words at program execution.

The second set is RPPy-specific:

<0	(n -- n) ZF = 1 if n < 0
>0	(n -- n) ZF = 1 if n > 0
=0	(n -- n) ZF = 1 if n == 0
!=0	(n -- n) ZF = 1 if n != 0
<0>	(idx1 idx2 idx3 n --) if n<0 execute word with idx1; if n=0 execute idx2; else idx3
<=>	(idx1 idx2 idx3 x y --) if x<y execute word with idx1; if x=y execute idx2, else idx3
zf=	(n --) pop TOS to ZF
zf	(-- ZF) push ZF

The ZF register can be accessed in the same manner as the index registers; you can store any value to be used by the branching words, always knowing that any non-zero value is equivalent to ZF == 1

Branching

The most common branching word is **if** with the following syntax: *"condition IF true part THEN false part"*, opposed to Python where you have:

```
IF condition:
  true part
ELSE:       # the ELSE is optional
  false part
```

In Python:

```
---
>>> a=10
>>> if a %2 == 0 :
...       print('Even number')
```

```
... else:
...       print('Odd number')
...
Even number
>>>
```

In RPPy there's no ELIF or ELSE:

```
ex> evenodd: 2 % =0 if " Even number" print ; then " Odd number" print ; .
 Data stack empty
ex> 10 evenodd .
Even number
 Data stack items: 1
[0]
ex> 11 evenodd .
Odd number
 Data stack items: 2
[0, 1]
ex>
```

There's also the opposite test, **ifz**, which executes the conditional part if ZF = 0, and the non-equality test **ifneq** which executes the conditional part if TOS is not equal to NOS. See **pcomp** which shows RPPy's compiling words.

```
ex> pcomp .
          ===Compiler words===
  #             ( -- ) comment, skip rest of line
  .             ( -- ) mark end of compile phase, start execution phase
  ""            ( -- ) mark start of docstring to embed in definition
  """           ( -- ) mark start of multiline string
  "             ( -- ) mark start of string with blanks and without double quotes
  ->            ( -- ) mark start of string with blanks/quotes of all sort
  ;             ( -- ) compile return or compile jump (if tail call optimisation)
  if            ( -- ) compile if: continue execution if ZF == 1, else branch to "then"
  ifz           ( -- ) compile ifz: same as if, but for ZF == 0
  ifneq         ( x y -- x y ) compile ifneq: continue execution if x!=y, else branch to
                "then"
  then          ( -- ) compile then: branch there if condition not satisfied

 Compiler Definitions:  11
 Data stack empty
ex>
```

Another branching word is **choose** with the following syntax:
" *index_of_word_for_true_branch index_of_word_for_false_branch CHOOSE* "
Let's redefine our "evenodd":

```
---
ex> even: " Even number" print ;
co> odd: " Odd number" print ;
co> evenodd1: 2 % =0 *even *odd choose ; .
 Data stack empty
ex> 10 evenodd1 .
Even number
 Data stack items: 1
[0]
ex> 13 evenodd1 .
Odd number
 Data stack items: 2
[0, 1]
ex>
---
```

Finally, there are two triple-branching words **<0>** and **<=>** which test for <0, =0, >0 and x<y, x=y, x>y respectively:

```
---
ex> less-0: " Negative number" print ;
co> equal-0: " Zero" print ;
co> greater-0: " Positive number" print ;
co> arglist: *less-0 *equal-0 *greater-0 ;
co> 3test: <0> ; .
 Data stack empty
ex> arglist -22 3test .
Negative number
 Data stack empty
ex> arglist 0 3test .
Zero
 Data stack empty
ex> arglist 99 3test .
Positive number
 Data stack empty
ex>
---
```

As a matter of convenience, put the indexes in a separate word to avoid stack shuffling, as working with three or more stack items becomes quickly cumbersome and error prone (not that with two items it's error free…)

Jumping

Of course there's no GO TO in RPPy, but any call to a word followed by a return is compiled as a single jump to that word. Useful in while-looping constructs, as in a sequence of the form:

word: w1 w2 w3 … *exit condition* if … ; then w4 w5 … word ;

where w1 w2 etc. are repeated until the exit condition is true. If there's no exit condition, the loop repeats indefinitely.

Vectored execution (indirect call)

The index of a word is similar to a function pointer in other languages; you can call a word directly by mentioning that word, or you can get the word's index with **name* and use **execidx** afterwards to call it. By storing the index in a variable you can:

- circumvent the fact that there is no forward reference possible in RPPy: store initially the index of a no-operation word in the definition, making it usable to test with this stub, then at a later stage define what you wanted to insert instead of the no-operation and replace it's index with the new one:

```
ex> pass: ;                          # no operation
co> vector: *pass execidx ;          # variable with initial no-op stored
co> w1: 11 22 vector " sum of 11 and 22" print ; . # word to test
 Data stack empty
ex> w1 .
sum of 11 and 22              # only message , without sum effectuated
 Data stack items: 2
[11, 22]
.
.
.
ex> plus: + ; .               # define later the real word which should replace no-
                              operation (remember that you cannot index a kernel
                              primitive directly, put it into a definition)
 Data stack items: 2
[11, 22]
ex> *plus *vector = .         # store index of real word into vector
 Data stack items: 2
[11, 22]
ex>w1 .                       # test now the same w1 word: sum is present on stack
sum of 11 and 22              # beware that w1 was not modified/edited, only the
 Data stack items: 3          # execution vector changed
[11, 22, 33]
```

- debug a word by storing data stack printing or any other relevant information printing in the execution vector, and replace it finally with a no-operation:

```
ex> prtstck: pds ;                   # print data stack
co> w2: 10 20 vector + 15 / vector ; .# insert printing where judged necessary
 Data stack empty
ex> *prtstck *vector = .             # set vector to printing
 Data stack empty
ex> w2 .                             # test w2 with stack printing
 Data stack items: 2
[10, 20]
 Data stack items: 1
[2.0]
 Data stack items: 1
[2.0]
```

```
ex> *pass *vector = .          # now set vector to no operation
 Data stack items: 1
[2.0]
ex> w2 .                       # w2 without stack printing
 Data stack items: 2
[2.0, 2.0]
ex>
```

See also “Switching” with regard to set a vector of multiple indexes.

Looping

A while loop executes as long as the while condition is true; in Python you have:

```
>>> a=10
>>> while a != 0 :
...         print(a)
...         a -= 1
...
10
9
8
7
6
5
4
3
2
1
>>>
```

In RPPy you have:

```
ex> while-true-loop: !=0 if dup print -- while-true-loop ; then " while ended"
print ; .
 Data stack empty
ex> 10 while-true-loop .
10
9
8
7
6
5
4
3
2
1
while ended
 Data stack items: 1
```

```
[0]
ex>
---
---
ex> while-false-loop: =0 if " while ended" print ; then dup print -- while-
false-loop ; .
 Data stack items: 1
[0]
ex> 10 while-false-loop .
10
9
8
7
6
5
4
3
2
1
while ended
 Data stack items: 2
[0, 0]
ex>
---
```

A counted loop uses in Python the range function:

```
---
>>> for a in range(1,11):
...   print(a)
...
1
2
3
4
5
6
7
8
9
10
---
```

In RPPy the counted loop uses an *index register* **I, J, K,** where the count is assigned; the body of the loop must be a separate word, repeatedly executed by **iloop**, **jloop** or **kloop**:

```
---
ex> iprint: i print i-- ;
co> counted-loop: i= *iprint iloop ; .
 Data stack empty
ex> 10 counted-loop .
10
```

```
9
8
7
6
5
4
3
2
1
 Data stack empty
ex>
---
```
  Beware that the word defining the body must be prefixed with "*",
to obtain it's index, as needed by i-j-k-loop.

Switching

  There are no predefined switch primitives in RPPy, but by using a
list of indexed words you can define either a *case-like numerical
switch* or a *key-driven* one:

```
---
ex> sw0: " case 0" print ;
co> sw1: " case 1" print ;
co> sw2: " case 2" print ;
co> sw-list: *sw0 *sw1 *sw2 ; .
ex> number-switch: "" ( switchlist n -- ) switch to n-th word in switchlist""
co> 2dup          # ( switchlist n switchlist n)
co> swap          # ( switchlist n n switchlist )
co> len --        # ( switchlist n n listlenght-1 )
co> > if " switch number out of list range" print ;
co> then 2drop   # ( switchlist n)
co> i= s[i]       # ( switchlist n-th_item_in_switchlist )
co> execidx drop      # ( ) execute word at n-th position in switchlist
co> ; .
ex> sw-list 2 number-switch .
case 2
 Data stack empty
ex> sw-list 0 number-switch .
case 0
 Data stack empty
ex> sw-list 3 number-switch .
switch number out of list range
 Data stack items: 4     # at n > listlenght no stack clearing is done,
[[12, 16, 20], 3, 3, 2]  # in order to identify faulty arguments
ex>                      # by the way, this definition has a big problem
---                      # even if input is ok (see drop after execidx...)
```

  Use a similar way for a key-driven switch; create a dictionary from two

lists: the key list and the associated indexed word list; execute the word found at key:index pair:

```
ex> keylist: ['first','second','third'] ; .
ex> keydict: keylist sw-list dictcre ; .
ex> keydict .
 Data stack items: 1          # see the indexes associated to keys in this case
[{'first': 12, 'second': 16, 'third': 20}]
ex> key-switch: "" ( keydict key -- ) switch to key:word association""
co> in ifz " key not found" print ;       # test if key present in dictionary
co> then get execidx       # get the index associated to key and execute word
co> ; .
ex> cleards
ex>
 Data stack empty
ex> keydict 'second key-switch .
case 1
 Data stack empty
ex> keydict 'first key-switch .
case 0
 Data stack empty
ex> keydict 'last key-switch .
key not found
 Data stack items: 2
[{'first': 12, 'second': 16, 'third': 20}, 'last']
ex>
```

You can even complete the above definitions by adding “out of range” or “key not found” treatment instead of error exit.

...NB. what's the problem with "drop" in number-switch above:
- if the called word leaves a result on the stack, drop drops it
- if we don't put a drop after execidx, the switchlist rests as garbage on the stack; decide then what to discard depending on execution flow

Pseudo-quotations (lambda/anonymous definitions)

By putting a set of indexed words in a list it’s possible to execute that list of words not only as above in the switching examples but integrally from start to end:

```
ex> w1: " starting execution of quotation" print ;
co> w2: " Transform an input string to all upper case: " inputprompt ;
co> w3: upper print ;
co> w4: " ending execution of quotation" print ;
co> x-list: 0 ; .          # define a list to store desired indexes
ex> pdefall .
     ...                   # see indexes
  w1        at index:  290 ""  ""
  w2        at index:  294 ""  ""
  w3        at index:  298 ""  ""
```

```
  w4        at index:  302 ""  ""
    ...
ex> '[] *w1 *w2 *w3 *w4 listcre .      # create index list
  Data stack items: 1
[[290, 294, 298, 302]]
ex> reverse .           # reverse it, as popping works from listend
  Data stack items: 1   # or, store in reverse order w4, w3, … if it's more pleasuring...
[[302, 298, 294, 290]]
ex> *x-list = .         # store reversed list in the execution vector
 Data stack empty
ex> x-loop: x-list pop nip execidx ; .   # pop an index and execute corresponding
                                         word
ex> exec-list: x-list len i= *x-loop iloop ; .# loop with I=nb of indexes
 Data stack empty
ex> exec-list .         # execute vector content
starting execution of quotation
Transform an input string to all upper case: abcd efgh
ABCD EFGH
ending execution of quotation
 Data stack empty
ex>
---
```

NB. Why pseudo-quotation and not quotation? Unfortunately you can't index compiler words, as if, then, string markers, etc., as RPPy doesn't compile at execution time. Therefore you can't define any desired word as a series of indexes, as it would be in a classic lambda function. Sorry guys...

Input and Printing

The word **input** reads a line from standard input, converts it to a string (stripping a trailing newline), and pushes the string to the data stack:

input (-- str) read a line from input and convert data to a string

```
---
ex> input .
Hello World!
 Data stack items: 1
['Hello World!']
ex>
---~
```

Do not attempt to input numbers and then use math on them directly, as the stack contains strings, not numbers:

```
---
ex> input .
50
 Data stack items: 2
['Hello World!', '50']
ex> input .
10
 Data stack items: 3
['Hello World!', '50', '10']
ex> + .                    # the "+" adds two strings here, not numbers
 Data stack items: 2    # convert them with int, float if math wanted
['Hello World!', '5010']
ex>
---
```

If you want a prompt before input, use **inputprompt**:

inputprompt (strprompt -- str) write strprompt to standard output, then read a line

```
---
ex> " Enter your name here: " inputprompt .
Enter your name here: Max Weber
 Data stack items: 1
['Max Weber']
ex>
---
```

You have already used some indispensable printing words like **pkern**, **pdef**, etc. , but let's present all RPPy specific printing words:

- **pds**: print data stack; useful if you switch the automatic stack printing off in order to unclutter console output
- **pdson**/**pdsoff**: switch on/off the data stack printing; at RPPy start, the switch is on

NB. if the data stack contains more than 10 items, *only the top 10*

will be displayed; use **cleards** to empty accumulated unnecessary data
---
```
ex> cleards .
 Data stack empty
ex> 11 22 33 .
 Data stack items: 3
[11, 22, 33]
ex> pdsoff .
ex> 44 55 .
ex> + .
ex> pds .
 Data stack items: 4
[11, 22, 33, 99]
ex> drop .
ex> swap + .
ex> pdson .
 Data stack items: 2
[11, 55]
ex>
```

- **pus**: print user stack; as seen in "Constants, variables ..." the user stack holds temporary data, avoiding the need of locals in words. Use **uspush** and **uspop** to transfer/retrieve data; **clearus** clears the user stack similarly to **cleards**

```
ex> cleards .
 Data stack empty
ex> 88 99 .
 Data stack items: 2
[88, 99]
ex> uspush .
 Data stack items: 1
[88]
ex> uspush .
 Data stack empty
ex> pus .
 User stack items: 2
[99, 88]
 Data stack empty
ex> uspop pus .
 User stack items: 1
[99]
 Data stack items: 1
[88]
ex> uspop pus .
 User stack empty
 Data stack items: 2
[88, 99]
ex>
```

- **prs**: print return stack; see the chain of words called
NB1. the return stack can't be modified by the user, only visualized
NB2. at abort, the return stack is always displayed, then emptied
---
```
ex> prs .
 Return stack items:0
 Data stack empty
ex> xx: " xx executed" print prs ;
co> yy: " yy executed" print xx ;
co> zz: " zz executed" print yy ;
co> aa: zz yy xx ; .
 Data stack empty
ex> aa .
zz executed
yy executed
xx executed
 Return stack items:2
2 return from: zz
1 return from: aa
yy executed
xx executed
 Return stack items:2
2 return from: yy
1 return from: aa
xx executed
 Return stack items:1
1 return from: aa
 Data stack empty
ex> prs .
 Return stack items:0
 Data stack empty
ex>
```

- **plist**: print list as enumerated items

```
ex> ['one','two','three'] .
 Data stack items: 1
[['one', 'two', 'three']]
ex> plist .
   0 one
   1 two
   2 three
 Data stack empty
ex>
```

- **pexlst**: print Execution List starting at given index; useful if execution aborted by various reasons, to see the execution stream where it happened
---

```
ex> " some work before declaring a definition" .
 Data stack items: 1
['some work before declaring a definition']
ex> print .
some work before declaring a definition
 Data stack empty
ex> 22 33 * 3 / .
 Data stack items: 1
[242.0]
ex> dup 5 // .
 Data stack items: 2
[242.0, 48.0]
ex> 1 pexlst .
    1 ;               # starting at index 2 you see the execution history
    2 lit  some work before declaring a definition
    3 print
    4 lit  22
    5 lit  33
    6 *
    7 lit  3
    8 /
    9 dup
   10 lit  5
   11 //
   12 lit  1          # the last thing done is calling pexlst
   13 pexlst
 Data stack items: 2
[242.0, 48.0]
ex>
```

Now suppose we start defining some words:

```
ex> xx: " empty def" ;
co> yy: " call xx" xx ;
co> zz: yy print print ; .
 Data stack items: 2
[242.0, 48.0]
ex> cleards 888 999 .
 Data stack items: 2
[888, 999]
ex> 1 pexlst .
    1 ;                   # all the previous history is gone
    2 xx:                 # new definitions are stored always after the last
    3 lit  empty def      # one always here, ending with";" or jump
    4 ;                   # that way no execution stream remains stored, as
    5 yy:                 # it is the VOLATILE part; only the last exec-stream
    6 lit  call xx        # rests after the last def declared, to be also erased
    7 jump xx             # if starting the next definition (see after index 13)
    8 zz:
    9 call yy
   10 print
```

```
   11 print
   12 ;
   13 cleards
   14 lit  888
   15 lit  999
   16 lit  1
   17 pexlst
 Data stack items: 2
[888, 999]
ex> aa: " will be stored at index 13 after zz" print ; .
 Data stack items: 2
[888, 999]
ex> 8 pexlst .
    8 zz:
    9 call yy
   10 print
   11 print
   12 ;
   13 aa:
   14 lit  will be stored at index 13 after zz
   15 print
   16 ;
   17 lit  8
   18 pexlst
 Data stack items: 2
[888, 999]
ex>
---
```

- **print**: print item from the stack, item may be a number, string, list, dictionary, etc. (item --)
Contrary to Python, only one item can be printed at a time; also the output is only stdout, there's no printing to file (yet...)

```
---
ex> " string print" print .
string print
 Data stack empty
ex> 3.14 print .
3.14
 Data stack empty
ex> {'one':1,'two':2} print .
{'one': 1, 'two': 2}
 Data stack empty
ex> ('aaa',111,'bbb',222) print .
('aaa', 111, 'bbb', 222)
 Data stack empty
ex>
---
```

Each item is printed on a *new line*; use **printend** with desired separator if printing on the same line:
printend (item endstr --)

```
---
ex> " line 1" print " line 2" print " line 3" print .
line 1
line 2
line 3
 Data stack empty
ex> " line 1" " , " printend " line 2" " , " printend " line 3" print .
line 1, line 2, line 3
 Data stack empty
ex>
---
```
Use **printnl** to print a single newline
```
---
ex> " line 1" print printnl " line 2" print printnl .
line 1

line 2

 Data stack empty
ex>
---
```
If you want to print multiple items, use a list:
```
---
ex> p-item: pop " , " printend ;
co> multi-print: dup len i= *p-item iloop ; .
 Data stack empty
ex> [1,'abc','efg',2] multi-print .
2, efg, abc, 1,  Data stack items: 1
[[]]                    # as pop starts from the last item, the
ex> drop .              # output is last-to-first, so use reverse
 Data stack empty
ex> [1,'abc','efg',2] reverse multi-print drop .
1, abc, efg, 2,  Data stack empty
ex>
---
```
Finally, use **printf** for *formatted output*; push the format string followed by the item to print : (formatstr item --).
```
---
ex> formdec: " {:d}" ;
co> formcomma: " {:,d}" ;
co> formcentered: " {:^15,d}" ;
co> formpadded: " {:*^15,d}" ;
co> formfloat: " {:*^15.2f}" ;
co> formhex: " {:*>15X}" ; .
 Data stack empty
ex> formdec 9999 printf .
9999
 Data stack empty
ex> formcomma 9999 printf .
9,999
```

```
 Data stack empty
ex> formcentered 9999 printf .
    9,999
 Data stack empty
ex> formpadded 9999 printf .
*****9,999*****
 Data stack empty
ex> formfloat 9999 printf .
****9999.00****
 Data stack empty
ex> formhex 65535 printf .
**********FFFF
 Data stack empty
ex>
```

The formatting specification follows this form:
[[fill] align][sign][#][0][width][,][.precision][type] where:

fill: fill character for data too small to fit within the assigned space
align: **<** left ; **>** right ; **^** centered ; **=** justified
sign: **+** positive numbers have a plus sign and negative a minus sign
- negative numbers have a minus sign
(space) positive numbers preceded by space, negative have a minus sign
#: alternative format for numbers, for ex. hexadecimal nb. preceded by **0x**
0: output should be sign aware and padded with zeros
width: full width of data field (even if data won't fit in)
,: numeric data should have commas as thousands separator
.precision: numbers of characters after decimal point
type: output type
s or nothing: string
integer: **b** binary; **c** character; **d** decimal; **o** octal; **x** hex with lowercase;
X hex with uppercase; **n** locale-specific thousands separator
floating point: **%** percentage; **e** exponent 'e'; **E** exponent 'E'; **f** lowercase
fixed point; **F** uppercase fixed point; **g** lowercase general format;
G uppercase general format; **n** locale-specific general format

All elements should appear in the *correct order*, or an error is emitted; don't for example specify alignment before fill character.

NB1. As the print word in RPPy is single-item oriented, you cannot use numbered fields as in Python: form=" A {0} {1} and a {0} {2}" ; print(form.format("good", "day", "night")) won't work in RPPy (you'll get an index error here)

NB2. For the same reason *f-strings* are not defined in RPPy

File I/O

Working with files involves three steps:
- open file
- read/write from/to file
- close file

To **open** a file, push two strings: filename and mode of file use:
(filename filemode -- filehandle)
If only the filename is given, the *current file directory* is taken into account; you can also specify a relative or absolute filepath as *C:/main/Python/filename.ext* in Windows; *do not use backslash* as folder separator as usually in Windows; RPPy uses only forward slash here.
The file mode has following values:
'r open for reading
'w open for writing, truncating the file first
'x open for exclusive creation, failing if the file already exists
'a open for writing, appending to the end of file if it exists
'b binary mode
't text mode
'+ open for updating (reading and writing)

---
ex> **'testfile.txt 'w open .**
Data stack items: 1
[<_io.TextIOWrapper name='testfile.txt' mode='w' encoding='cp1252'>]
ex>

What you get is a file object, called *filehandle* in RPPy, used in all consequent operations on this file.
Now define a text and write it to testfile.txt:
---
ex> **filetext: """**
""> **line 1**
""> **line 2**
""> **line 3**
""> **eot**
""> **"""**
Compile warning: all data after closing triple-quote in current line is ignored!
co> **; .**
Empty line, nothing to execute
Data stack items: 1
[<_io.TextIOWrapper name='testfile.txt' mode='w' encoding='cp1252'>]
ex> **dup uspush .** # save filehandle to user stack, as it'll be needed at close
Data stack items: 1
[<_io.TextIOWrapper name='testfile.txt' mode='w' encoding='cp1252'>]
ex> **filetext .**

```
 Data stack items: 2
[<_io.TextIOWrapper name='testfile.txt' mode='w' encoding='cp1252'>, ' \nline
1\nline 2\nline 3\neot\n']
ex> write .        # write needs the filehandle followed by the textstring to write
 Data stack items: 1
[27]               # after a succesfull write op, the number of written chars is pushed
ex> uspop .       # beware that \n counts as a single char: newline
 Data stack items: 2
[27, <_io.TextIOWrapper name='testfile.txt' mode='w' encoding='cp1252'>]
ex> close .        # always close file after writing ops, otherwise the operating
                        #system's buffering algorithm doesn't guarantee instant write;
 Data stack items: 1     # only when exiting RPPy all open remainded files are
                         # automatically closed
[27]
ex> drop .
 Data stack empty
ex>
---
```

Now **read** the file back and display it:

```
---
ex> 'C:/main/python/testfile.txt 'r open .  # try with absolute path
 Data stack items: 1
[<_io.TextIOWrapper name='C:/main/python/testfile.txt' mode='r'
encoding='cp1252'>]
ex> dup uspush .       # save filehandle as needed further by close
 Data stack items: 1
[<_io.TextIOWrapper name='C:/main/python/testfile.txt' mode='r'
encoding='cp1252'>]
ex> read .
 Data stack items: 1
[' \nline 1\nline 2\nline 3\neot\n']
ex> print .

line 1
line 2
line 3
eot

 Data stack empty
ex> uspop close .      # get back filehandle for close
 Data stack empty
ex>
---
```

NB1. Unfortunately RPPy has no Python-like *contextual coding* using "with", which assures automated file close at end of operations; this forces you to *manually close* files as shown above.

NB2. Open works in RPPy with following keyword arguments frozen: buffering=-1, encoding=None, errors=None, newline=None, closefd=True,

opener=None ; only filename and filemode are taken into account.
Also modes "r" and "t" which are default on Python, must always be specified

NB3. All examples are *validated for Windows*, working with paths is a
little different in Linux or MacOS and not treated here

JSON Words

JSON (JavaScript Object Notation) is a lightweight data interchange format inspired by JavaScript object literal syntax. Converting between JSON and Python objects is done as follows:

Python	**JSON**
dict	object
list	array
tuple	array
str	string
int,float	number
True	true
False	false
None	null

Convert a Python object to JSON string with **jsdumps** and from JSON back to Python with **jsloads** :

```
ex> {'Name':'Jeff','age':31} .
 Data stack items: 1
[{'Name': 'Jeff', 'age': 31}]
ex> jsdumps .          # convert a dictionary to JSON
 Data stack items: 1
['{"Name": "Jeff", "age": 31}']
ex> jsloads .          # convert back from JSON
 Data stack items: 1
[{'Name': 'Jeff', 'age': 31}]
ex>
```

Beware that if a key in key:value pair is a *number*, it is converted to a *string* in JSON, so at reconversion back to Python dictionary *the key remains a string*, not number as in the original dictionary:

```
ex> {111:'key1',222:'key2'} update .
 Data stack items: 1
[{'Name': 'Jeff', 'age': 31, 111: 'key1', 222: 'key2'}]
ex> jsdumps .
 Data stack items: 1
['{"Name": "Jeff", "age": 31, "111": "key1", "222": "key2"}']
ex> jsloads .          # the keys 111 and 222 are strings now
 Data stack items: 1
```

```
[{'Name': 'Jeff', 'age': 31, '111': 'key1', '222': 'key2'}]
ex>
---
```

Similar to **jsdumps** and **jsloads** there are:
jsdump (filename pyobj --) write Python object as JSON object to filename
jsload (filename -- pyobj) load JSON object as Python object from filename

```
---
ex> cleards
ex>
 Data stack empty
ex> 'testjs.jss {'Monday':1,'Tuesday':2} jsdump .
 Data stack empty
ex> 'testjs.jss jsload .
 Data stack items: 1
[{'Monday': 1, 'Tuesday': 2}]
ex>
---
```

All following parameters, except filename and object/string are frozen in RPPy:
jsload/jsloads:
(cls=None, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, object_pairs_hook=None)
jsdump/jsdumps:
(skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, default=None, sort_keys=False)

See Python docs for relevant information concerning above keyword parameters.

Miscellaneous Words

- **abort** (str --) print abort message str, switch to REPL
Useful to abort execution if a certain condition is not satisfied:

```
---
ex> testneg:  <0 if " negative value not allowed" abort
co> then " value ok" print drop ; .
 Data stack empty
ex> 33 testneg .
value ok
 Data stack empty
ex> -33 testneg .
User Abort
"abort" aborted: negative value not allowed
 Aborted at Execution List Index:  6
 In definition: "testneg" at index: 2
  TOS:  <class 'int'> -33
 Return stack items:1
1 return from: testneg
 Data stack items: 1
[-33]
ex>
---
```

- **all** (seq --) ZF=1 if all elements in seq are true or seq is empty

```
---
ex> testall: all if " all elements are true or empty sequence" print ;
co> then " at least one element is false" print ; .
 Data stack empty
ex> [1,2,3,4] testall .
all elements are true or empty sequence
 Data stack empty
ex> emptystr testall .
all elements are true or empty sequence
 Data stack empty
ex> [] testall .
all elements are true or empty sequence
 Data stack empty
ex> [1,2,0,4] testall .
at least one element is false
 Data stack empty
ex>
---
```

- **any** (seq --) ZF=1 if any element in seq is true; ZF=0 if seq is empty

```
---
ex> testany: any if " at least one element is true" print ;
co> then " sequence is empty" print ; .
 Data stack empty
```

```
ex> [1,0,3,0] testany .
at least one element is true
 Data stack empty
ex> [] testany .
sequence is empty
 Data stack empty
ex>
---

  - bin ( n -- strbin ) convert integer n to binary string
---
ex> 255 bin print .
0b11111111
 Data stack empty
ex>
---

  - chr ( n -- strchr ) convert integer n to string representing associated glyph
---
ex> 65 chr print .
A
 Data stack empty
ex> 17 chr print .
◄
 Data stack empty
ex> 17 chr .
 Data stack items: 1
['\x11']
ex>
---

  - complex ( str -- n ) convert str to complex number n
---
ex> '11+22j complex .
 Data stack items: 1
[(11+22j)]
ex> '33 complex .
 Data stack items: 2
[(11+22j), (33+0j)]
ex> '44j complex .
 Data stack items: 3
[(11+22j), (33+0j), 44j]
ex>
---

  - choose ( idx1 idx2 -- ) if ZF=1 execute word with idx1, else idx2
  See "Control Flow"

  - enumerate ( seq -- list ) list of tuples (count,value) iterating over seq
---
ex> 'abcd enumerate .
```

```
 Data stack items: 1
[[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd')]]
ex>
---
---
ex> {'Fred':32,'Anna':24,'Gregory':40} enumerate .
 Data stack items: 1
[[(0, 'Fred'), (1, 'Anna'), (2, 'Gregory')]]
ex>
---
```

- **eval** (str -- item) TOS = eval(str)

```
---
ex> " (10 + 20) * 3 / 2" eval .
 Data stack items: 1
[45.0]
ex> (10+20)*3/2 .          # if no spaces, the same expression, without
 Data stack items: 2       # string and eval, is evaluated as above, but
[45.0, 45.0]               # here you must use INFIX notation, with regard
ex> 10 20 + 3 * 2 / .      # to Python's precedence rules
 Data stack items: 3       # ... and now in postfix
[45.0, 45.0, 45.0]
ex> 2 3 10 20 + * swap / .
 Data stack items: 4
[45.0, 45.0, 45.0, 45.0]
ex>
---
```

NB. RPPy uses Python's **eval** as input in the Read-Eval-Print-Loop, as consequence any valid Python expression containing *only number/string values* can be evaluated, using *infix notation* and known *precedence rules*. But using variables, as in Python, doesn't work, a=10; b=20; eval("a+b") is a valid Python expression and a rejected RPPy one:

```
---
ex> a: 10 ;
co> b: 20 ; .
 Data stack empty
ex> " a+b" eval .
"eval" aborted: name 'a' is not defined
 Aborted at Execution List Index:  48
 Aborted in execution string or in definition with multiple returns
  TOS:  <class 'str'> a+b
 Data stack items: 1
['a+b']
ex>
---
```

The variable "a" is defined as a word, not as a Python variable, so it is not visible to the **eval** function. But you can define Python variables, see **exec**

- **exec** (str -- ...) exec(str), stack depends of what exec does

```
---
ex> pyvar: """
""> a=10
""> b=20
""> c=eval("a+b")
""> print("Executed as Python statements")
""> print("c=",c)
""> """
Compile warning: all data after closing triple-quote in current line is ignored!
co> ; .
 Data stack empty
ex> pyvar exec .
Executed as Python statements
c= 30
 Data stack empty
ex>
---
```

Exec is a useful word if you want to *extend the capabilities of RPPy*; see more on this in "Extending RPPy".

- **execidx** (idx --) execute word with index idx

```
---
ex> exidx: " exidx executed with index: " "  " printend print ;
co> testidx: *exidx dup execidx ; .
 Data stack empty
ex> testidx .
exidx executed with index:  53
 Data stack empty
ex> 'exidx pdef .
  exidx       at index:  53 ""  ""
  exidx: " exidx executed with index: " "  " printend print ;

  No duplicate (older definitions) present
 Data stack empty
ex>
---
```

- **float** (str/int -- n) convert string or integer to floating point number

```
---
ex> " 3.151596 " float .
 Data stack items: 1
[3.151596]
ex> 99 float .
 Data stack items: 2
[3.151596, 99.0]
ex>
---
```

- **format** (formstr n -- str) convert value n to str according to formstr

```
---
ex> '{:*^10.3f} 99 format .
```

```
 Data stack items: 1
['**99.000**']
ex>
---
```
  See more **format** string examples in "Input and printing" for "printf"

  - **help** (--) print help chapters

  - **hex** (n -- strhex) convert integer n to hexadecimal string
```
---
ex> 65535 hex .
 Data stack items: 1
['0xffff']
ex>
---
```

  - **input** (-- str) read a line from input and convert data to a string
  See "Input and printing"

  - **inputprompt** (strprompt -- str) write strprompt to standard output,
  then read a line
  See "Input and printing"

  - **int** (str/n -- n) convert str or number n to integer n
```
---
ex> 3.14 int .
 Data stack items: 1
[3]
ex> " 99" int .
 Data stack items: 2
[3, 99]
ex>
---
```

  - **intbase** (str base -- n) convert str to integer n according to base
```
---
ex> '999 10 intbase .
 Data stack items: 1
[999]
ex> 'ffff 16 intbase .
 Data stack items: 2
[999, 65535]
ex> '1100 2 intbase .
 Data stack items: 3
[999, 65535, 12]
ex> 'abcd 36 intbase .
 Data stack items: 4
[999, 65535, 12, 481261]
ex>
---
```

- **intro** (--) print introduction to RPPy

- **license** (--) print RPPy license

- **oct** (n -- stroct) convert integer n to octal string
---
ex> **63 oct .**
Data stack items: 1
['0o77']
ex> **64 oct .**
Data stack items: 2
['0o77', '0o100']
ex>

- **ord** (strchr -- n) convert string representing one character to integer n
---
ex> **'A ord .**
Data stack items: 1
[65]
ex> **'a ord .**
Data stack items: 2
[65, 97]
ex> **" " ord .**
Data stack items: 3
[65, 97, 32]
ex>

- **quit** (... -- ...) end execution of RPPy
Works similarly to **Ctrl-Q**, asking for definitions to **save**, to prevent accidentally loosing what has been defined in a work session.

- **str** (item -- str) return a string interpretation of item
---
ex> **99.998 str .**
Data stack items: 1
['99.998']
ex> **[1,'abcd',2] str .**
Data stack items: 2
['99.998', "[1, 'abcd', 2]"]
ex>

- **type** (item -- itemtype) TOS = type of item
---
ex> **{} type .**
Data stack items: 1
[<class 'dict'>]
ex> **(11+22j) type .**
Data stack items: 2

```
[<class 'dict'>, <class 'complex'>]
ex> (1,2,3) type .
 Data stack items: 3
[<class 'dict'>, <class 'complex'>, <class 'tuple'>]
ex> {1,2,3} type .
 Data stack items: 4
[<class 'dict'>, <class 'complex'>, <class 'tuple'>, <class 'set'>]
ex> [1,2,3] type .
 Data stack items: 5
[<class 'dict'>, <class 'complex'>, <class 'tuple'>, <class 'set'>, <class 'list'>]
ex>
```

NB. If you want to use the result of type in comparisons, either convert it to a string or generate the compared type with a known value:

```
ex> 123 type str " <class 'int'>" == if " Integer" print then .
Integer
 Data stack items: 2
["<class 'int'>", "<class 'int'>"]
ex> [1,2,3] type [] type == if " List " print then .
List
 Data stack items: 4
["<class 'int'>", "<class 'int'>", <class 'list'>, <class 'list'>]
ex>
```

A Few Examples

Factorial function

The factorial function is defined as:
0! = 1 ; 1! = 1; n! = n(n-1)!

```
ex> ff: "" ( n n -- n! 1 )"" 1 == drop if ;
co> then -- dup rot * swap ff ; .
# ( n n -- n n 1 )              1 ==     exit at terminating condition n==1
# ( -- n n )                    drop
# continue until n == 1         if
# ( n n -- n n-1 n-1 )          -- dup
# ( -- n-1 n-1 n )              rot
# ( -- n-1 n*(n-1) )            *
# ( -- n*(n-1) n-1 )            swap
# now TOS is n-1, jump back to beginning test , NOS gets n*(n-1)*(n-2)… = n!
 Data stack empty
ex> 6 6 ff .
 Data stack items: 2
[720, 1]
ex> n!: "" ( n  -- n!  )"" dup ff drop ; .
 Data stack items: 2
[720, 1]
ex> cleards .
 Data stack empty
ex> 6 n! .
 Data stack items: 1
[720]
ex> 10 n! .
 Data stack items: 2
[720, 3628800]
ex> 12 n! .
 Data stack items: 3
[720, 3628800, 479001600]
ex> 20 n! .
 Data stack items: 4
[720, 3628800, 479001600, 2432902008176640000]
ex>
```

Great, isn’t it? Try -6 n! , 0 n! , 6.123 n! (hint: save your work before …)
So define a little better factorial:

```
ex> N!: int =0 if drop 1 ;  # convert to integer, test if 0! demanded
co> then >0 if n! ;         # ok if positive number
co> then abs n! ; .         # absolute value if negative number
ex> 6 N! .
 Data stack items: 1
[720]
ex> 0 N! .                  # 0! is defined as 1 by convention for an empty product
```

```
 Data stack items: 2
[720, 1]
ex> -6 N! .             # cheated here, factorial is defined for positive numbers
 Data stack items: 3
[720, 1, 720]
ex> 6.1234 N! .         # cheated here, factorial is defined for integers
 Data stack items: 4
[720, 1, 720, 720]
ex>
---
  And a variant with messages without cheating:
---
ex> NN!: <0 if " Negative number" print drop ;
co> then =0 if drop 1 ;
co> then dup type str " <class 'int'>" == if 2drop n! ;
co> then " Not an integer" print drop 2drop ; .
ex> 6 NN! .
 Data stack items: 1
[720]
ex> 0 NN! .
 Data stack items: 2
[720, 1]
ex> -6 NN! .
Negative number
 Data stack items: 2
[720, 1]
ex> 6.123 NN! .
Not an integer
 Data stack items: 2
[720, 1]
ex>
---
  But even NN! isn’t perfect: wrong order of input testing...
---
ex> 'xx NN! .
"<0" aborted: '<' not supported between instances of 'str' and 'int'
 Aborted at Execution List Index:  107
 In definition: "NN!" at index: 106
  TOS:  <class 'str'> xx
  NOS:  <class 'int'> 1
 Return stack items:1
1 return from: NN!
 Data stack items: 3
[720, 1, 'xx']
ex>
---
  Hopefully this is at least OK...
---
ex> NN!: dup type str " <class 'int'>" == if
co> 2drop =0 if drop 1 ;
co> then <0 if " Negative number" print drop ;
```

```
co> then n! ;
co> then " Not an integer" print drop ; .
 Data stack empty
ex> 6 NN! .
 Data stack items: 1
[720]
ex> 0 NN! .
 Data stack items: 2
[720, 1]
ex> -6 NN! .
Negative number
 Data stack items: 2
[720, 1]
ex> 6.123 NN! .
Not an integer
 Data stack items: 4
[720, 1, 6.123, "<class 'float'>"]
ex> 'xxxxx NN! .
Not an integer
 Data stack items: 6
[720, 1, 6.123, "<class 'float'>", 'xxxxx', "<class 'str'>"]
ex>
```

Another idea is the use of **val*** which multiplies the top n stack items and an index register to generate n, n-1, n-2, n-3, … 1:

```
ex> n-loop: i-- i ;                  # decrement i and leave value on stack
co> x: dup i= *n-loop iloop ; . # repeat i times
 Data stack empty
ex> 5 x .
 Data stack items: 6
[5, 4, 3, 2, 1, 0]                   # discard the last zero, the rest is ok
ex> cleards .
 Data stack empty

# so far, so good: we have the number series generated; add their multiply part

ex> x!: dup uspush x drop uspop val* ; .
 Data stack empty
ex> 6 x! .
 Data stack items: 1
[720]
ex> 10 x! .
 Data stack items: 2
[720, 3628800]
ex> 1 x! .
 Data stack items: 3
[720, 3628800, 1]
```

Try adding tests for zero, negative or not integer input; otherwise following errors appear:

```
---
ex> 0 x! .
"val*" aborted: Nb. of values to multiply is > stacklenght, or < 1
 Aborted at Execution List Index:  273
 In definition: "x!" at index: 264
  TOS:  <class 'int'> 0
  NOS:  <class 'int'> 1
 Return stack items:1
1 return from: x!
 Data stack items: 4
[720, 3628800, 1, 0]
ex> -6 x! .
"val*" aborted: Nb. of values to multiply is > stacklenght, or < 1
 Aborted at Execution List Index:  273
 In definition: "x!" at index: 264
  TOS:  <class 'int'> -6
  NOS:  <class 'int'> 0
 Return stack items:1
1 return from: x!
 Data stack items: 5
[720, 3628800, 1, 0, -6]
ex> 'xxx x! .
"i=" aborted: Index value must be an integer
 Aborted at Execution List Index:  268
 In definition: "x!" at index: 264
  TOS:  <class 'str'> xxx
  NOS:  <class 'str'> xxx
 Return stack items:1
1 return from: x!
 Data stack items: 7
[720, 3628800, 1, 0, -6, 'xxx', 'xxx']
ex>
---
```

Fibonacci sequence

The Fibonacci sequence is a sequence in which each number is the sum of the two preceding ones. Starting from 1 and 1 it is:
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

```
---
ex> fibgen:       # generate one fibonacci number
co> 2dup          # ( x y -- x y x y )
co> +             # ( x y x y -- x y x+y )
co> "" ( x y -- x y x+y ) generate a fibonacci number ""
co> ; .
 Data stack empty
ex> 1 2 fibgen .  # start with 1 and 2 on the stack
 Data stack items: 3
[1, 2, 3]
```

```
ex> fibgen .
 Data stack items: 4
[1, 2, 3, 5]
ex> fibgen .        # ok, it works; include fibgen in a loop to create desired sequence
 Data stack items: 5
[1, 2, 3, 5, 8]
ex> fibo: "" ( n -- list of first n+2 fibonacci numbers)"" # n+2 as the first pair is
                                                  # already here
co> i= '[] 1 1 *fibgen iloop listcre ; . # set I to desired sequence length, push the
                                                  # "[]" list start marker, push initial pair 1 1
 Data stack empty
ex> 5 fibo .
 Data stack items: 1
[[1, 1, 2, 3, 5, 8, 13]]
ex> cleards .
 Data stack empty
ex> 8 fibo .
 Data stack items: 1
[[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]]
ex> cleards .
 Data stack empty
ex> 10 fibo .
 Data stack items: 1
[[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]]
ex>
---
```

Square root with Newton's method

You can always use math.sqrt() as in "Extending RPPy", but let's define it ourselves with Newton's formula:

y = (x+a/x) / 2

where the square root of *a* is calculated choosing *x* as a first estimate and continue with replacing the result *y* as a new estimate until *x* becomes enough close to *y* (being floating numbers, exact equality is not always obtainable)

```
---
ex> y: "" ( x a -- (x+a/x)/2 )""
co> over            # ( x a x )
co> /               # ( x a/x )
co> +               # ( x+a/x )
co> 2 /             # ( (x+a/x)/2 )
co> ; .
 Data stack empty
ex> # square root of 4 with first approx. x=3
ex> 3 4 y .
 Data stack items: 1
[2.1666666666666665]
ex> # continue with new approx. 2.16666...
ex> 4 y .
 Data stack items: 1
```

```
[2.0064102564102564]
ex> # not bad, continue
ex> 4 y .
 Data stack items: 1
[2.0000102400262145]
ex> 4 y .
 Data stack items: 1
[2.0000000000262146]
ex> 4 y .
 Data stack items: 1
[2.0]
ex> # GOT IT !
---
```

Now define a word which loops until two succesive estimates are close to an epsilon as difference between themselves; also display stack content at each iteration.

```
---
ex> sqroot: "" ( x a -- )""
co> 2dup uspush uspush      # ds( x a ) us( a x )  ds=data stack; us=user stack
co> y dup uspop             # ds( y y x ) us( a )
co> - abs                   # ds( y |y-x|) us( a )
co> 0.0000001 < if          # ds( y |y-x| 0.0000001 ) us( a ) epsilon=1e-7
co> 2drop                   # ds( y ) us( a )
co> print                   # ds( ) us( a )   print square root of a
co> uspop print ;           # ds( ) us( )      print a, exit
co> then 2drop uspop        # ds( y a ) us( ) reload a, restart with new estimate
co> pds sqroot ; .          # ds( y a ) us( ) print ds at each cycle
 Data stack empty
ex> 3 4 sqroot .
 Data stack items: 2
[2.1666666666666665, 4]
 Data stack items: 2
[2.0064102564102564, 4]
 Data stack items: 2
[2.0000102400262145, 4]
 Data stack items: 2
[2.0000000000262146, 4]
2.0
4
 Data stack empty
ex> 3 10 sqroot .
 Data stack items: 2
[3.166666666666667, 10]
 Data stack items: 2
[3.162280701754386, 10]
 Data stack items: 2
[3.162277660169842, 10]
3.162277660168379
10
 Data stack empty
ex> 60 225 sqroot .
 Data stack items: 2
```

```
[31.875, 225]
 Data stack items: 2
[19.466911764705884, 225]
 Data stack items: 2
[15.51249270954841, 225]
 Data stack items: 2
[15.008465717994458, 225]
 Data stack items: 2
[15.000002387598524, 225]
 Data stack items: 2
[15.00000000000019, 225]
15.0
225
 Data stack empty
ex>
```

Define a new variant where displaying the stack is optional using vectored execution (see "Control Flow – Vectored Execution" for details); also do not print final results, leave them on stack for further use; you'll see why...

```
ex> noop: ;                          # no operation
co> printstack: pds ;                # print data stack
co> switch: 0 execidx ;              # variable holding the execution index
co> yespds: *printstack *switch = ;    # set switch to printing stack
co> notpds: *noop *switch = ;          # set switch to no operation
co> sqroot.v2: "" ( x a – sqrt(a) a )""
co> 2dup uspush uspush         # same as sqroot, no comments added
co> y dup uspop - abs 1e-07 < if 2drop uspop ;
co> then 2drop uspop switch sqroot.v2 ; . # switch to print/not print data stack
ex> yespds 3 4 sqroot.v2 .     # data stack print enabled
 Data stack items: 2
[2.1666666666666665, 4]
 Data stack items: 2
[2.0064102564102564, 4]
 Data stack items: 2
[2.0000102400262145, 4]
 Data stack items: 2
[2.0000000000262146, 4]
 Data stack items: 2
[2.0, 4]
ex> notpds 3 4 sqroot.v2 .     # data stack print disabled
 Data stack items: 4
[2.0, 4, 2.0, 4]
ex>
```

Finally, add some bells and whistles: ask user for data and printing options:

```
# first define user dialogue to get x and a
ex> getvalues: "" ( -- x a ) get first estimate x & value to extract sqrt a ""
co> "  Input first estimate x: " inputprompt
# leave a string str(x)
```

```
co> dup len =0 if 2drop "  Exit, no first estimate done" ;
# zero length of delivered string answer means only Enter pressed
co> then drop int      # ( x )
co> "  Input value to extract square root a: " inputprompt
# leave a string str(a)
co> dup len =0 if 2drop drop "  Exit, no value to extract square root done" ;
co> then drop int ;    # ( x a ) integer values for x & a

# ask for printing options
ex> setswitch: "" ( -- ) set printing of succesive estimates on/off""
co> "  Print data stack with succesive estimates? y/n:" inputprompt
co> dup len =0 if 2drop *noop *switch = ; # no print if answer is Enter only
co> then drop upper 'Y == if 2drop *printstack *switch = ; # test for Y
co> then 2drop *noop *switch = ; . # no print if any  other value as input

# print result using printend to put output on the same line
ex> printsqrt: "" ( sqrt(a) a -- ) ""
co> "  Square root with Newton approximation of " "  " printend
co> "  " printend "  is: " "  " printend print ; .

# put all together into the final definition
ex> NewtonSquare: "" ( -- ) square root with user input""
co> getvalues     # ( x a -- )
co> setswitch     # ( x a -- )
co> sqroot.v2     # ( x a -- sqrt(a) a )
co> printsqrt     # ( sqrt(a) a -- )
co> ; .

ex> NewtonSquare .
 Input first estimate x: 60
 Input value to extract square root a: 225
 Print data stack with succesive estimates? y/n:y
 Data stack items: 2
[31.875, 225]
 Data stack items: 2
[19.466911764705884, 225]
 Data stack items: 2
[15.51249270954841, 225]
 Data stack items: 2
[15.008465717994458, 225]
 Data stack items: 2
[15.000002387598524, 225]
 Data stack items: 2
[15.00000000000019, 225]
 Square root with Newton approximation of  225  is:  15.0
 Data stack empty
```

NB. It works only for positive integers > 0; you can modify **getvalues** to get sqrt(0) = 0 , to work also for real numbers and to test for negative numbers (where the square root is a complex number).

Editing, Saving and Loading Definitions

There are two types of words in RPPy:
- *kernel primitives* written in Python
- *high level words* written as a combination of kernel primitives and other already defined high level words.

The kernel primitives are the equivalent of Python's built-in functions/methods; adding/modifying them can be done only by changing the RPPy script at source level.

Each time RPPy is launched, only the kernel primitives are available for use; the user has to create his own definitions, saving them for further reuse/completion.

Saving definitions is done by:
- **save** (filename --) save RPPy user generated words to filename

The filename must be *without extension,* as RPPy adds **.rpp** as extension; you can use absolute or relative path with filename, otherwise if only the name is given the file is created in the current directory.
- **Ctrl-S** or **s.** (--) save RPPy user generated words to file "*tempsave.rpp*"

The file "tempsave.rpp" is always created in the current directory; also **Ctrl-S** has to be pressed only at *start of line input*. Use **s.** as an alternative where **Ctrl-S** poses problems by the operating system.
If you exit RPPy with **Ctrl-Q** or **quit**, there's always the option of saving/not saving all definitions created in the current session.

Loading definitions is done by:
- **load** (filename --) load RPPy user generated words from filename

As with **save**, the filename must be *without extension*, as only files of type **.rpp** are recognized.
Loading *always overwrites all existing definitions*, if any, there's no concept of overlays in RPPy. You start at beginning with an empty dictionary, then create a first set of definitions; save them, and from now on, at each working session *load them first* and continue adding other definitions until finishing the application.
A very important issue about *variables* (which are also definitions): at **save**, always their *LAST VALUE* is written on file, from this point of view *RPPy works like a spreadsheet*: what you have in the cells at saving will be retrieved at next loading. Problems could arise if, as example, a variable is used to initialise a loop index and incremented afterwards; it'll be saved with the last count and not the starting one, so at next reloading the loop goes awry...

```
ex> pdefall .

 High Level Dictionary Definitions:  0
 Data stack empty
ex> w1: " first def" print ;
co> w2: " second def" print ;
```

```
co> w3: " third def" print ; .
 Data stack empty
ex> 'c:/main/python/test-save-load save .
Saved: 3 definitions to c:/main/python/test-save-load.rpp
 Data stack empty
ex> w4: 444 ;
co> w5: 555 ; .
 Data stack empty
ex> pdefall .
  w1        at index:  2 ""  ""
  w2        at index:  6 ""  ""
  w3        at index:  10 ""  ""
  w4        at index:  14 ""  ""
  w5        at index:  17 ""  ""

 High Level Dictionary Definitions:  5
 Data stack empty
ex> ^Q
Saved only 3 definitions from a total of 5
Quit anyway? (y/n):n
 Data stack empty
ex> ^S
Saved: 5 definitions to tempsave.rpp
 Data stack empty
---

  - deldef ( defname -- ) delete definition defname
---
ex> 'w1 deldef 'w2 deldef .
"w1" at index 2 deleted
"w2" at index 6 deleted
 Data stack empty
ex> pdefall .
  w3        at index:  10 ""  ""
  w4        at index:  14 ""  ""
  w5        at index:  17 ""  ""

 High Level Dictionary Definitions:  3
 Data stack empty
ex> 'c:/main/python/test-save-load load .
! Warning: all current definitions will be lost - proceed? (y/n): y
  Definitions loaded:
    0 w1
    1 w2
    2 w3
 Data stack empty
ex> pdefall .
  w1        at index:  2 ""  ""
  w2        at index:  6 ""  ""
  w3        at index:  10 ""  ""
```

```
 High Level Dictionary Definitions:  3
 Data stack empty
ex> 'tempsave load .
! Warning: all current definitions will be lost - proceed? (y/n): y
  Definitions loaded:
    0 w1
    1 w2
    2 w3
    3 w4
    4 w5
 Data stack empty
ex> pdefall .
  w1        at index:  2 ""  ""
  w2        at index:  6 ""  ""
  w3        at index:  10 ""  ""
  w4        at index:  14 ""  ""
  w5        at index:  17 ""  ""

 High Level Dictionary Definitions:  5
 Data stack empty
ex> ^Q
RPPy ended; last saved: 5 definitions
---
```

Editing a definition is done by:
- **edit** (defname --) edit definition defname

```
---
ex> w1: " Hello!" print ;
co> w2: w1 " How are you?" print ;
co> w3: w1 " Glad to see you" print ; .
 Data stack empty
ex> w2 .
Hello!
How are you?
 Data stack empty
ex> w3 .
Hello!
Glad to see you
 Data stack empty
ex> 'w1 edit .
  Warning: works only in Windows Command Prompt !
    Type Ctrl-M, cursor changes shape
    Use arrow keys to move cursor at beginning of displayed definition
    Hold Shift pressed, move right arrow to select entire definition
    Type Ctrl-Insert to copy selected text to clipboard, cursor moves back to
input line
    Type Ctrl-V to paste clipboard to input line
    Edit input line , terminate as usually with "." plus Enter
  w1        at index:  2 ""  ""
  w1: " Hello!" print ;

  No duplicate (older definitions) present
```

```
 Data stack empty
ex> w1: " Hello dear user!" print ; .  # new variant of w1
Empty line, nothing to execute
 Data stack empty
ex> w4: w1 " Good bye!" print ; .       # w4 uses the new w1 variant
 Data stack empty                       # because w4 is defined after new w1
ex> w4 .
Hello dear user!
Good bye!
 Data stack empty
ex> w2 .                # w2 and similarly w3 retain the old variant of w1
Hello!                  # if all occurencies of w1 should be replaced, use repdef
How are you?
 Data stack empty
---
```

- **repdef** (defname --) replace old definitions of defname with the last one defined

```
---
ex> 'w1 repdef .
  Replacing 1 older definitions
  Replaced in 2 definitions
 Data stack empty
ex> w2 .              # now all words use the last variant of w1
Hello dear user!
How are you?
 Data stack empty
ex> w3 .
Hello dear user!
Glad to see you
 Data stack empty
ex> w4 .
Hello dear user!
Good bye!
 Data stack empty
ex>
---
```

- **refdef** (defname -- list) list of all definitions including a reference to defname

```
---
ex> 'w1 refdef .
 Data stack items: 1
[['w2', 'w3', 'w4']]
ex>
---
```

Extending RPPy

Using **exec** you can extend the functionalities of RPPy:
- **exec** (str -- ...) exec(str), stack depends of what exec does

If for example you want to calculate the square root of a number, you can define a word which imports the math module:

```
---
ex> sqbase: """              # define a multiline string including all
""> import math              # desired Python statements to be executed
""> if tos() < 0 :
"">     print('Negative number')
""> else:
"">     dpush(math.sqrt(dpop()))
""> """
Compile warning: all data after closing triple-quote in current line is ignored!
co> ; .
 Data stack empty
ex> sqrt: "" ( n -- sqrt(n))""
co> sqbase exec ; .
 Data stack empty
ex> 144 sqrt .
 Data stack items: 1
[12.0]
ex> -144 sqrt .
Negative number
 Data stack items: 2
[12.0, -144]
ex>
---
```

But you need to know a little bit of RPPy internals to interface it with Python:
- use *dpush(xxx)* to push data on the data stack
- use *dpop()* to pop data from the data stack
- use *tos()* and *nos()* if you want to test values on TOS & NOS without discarding them
- use **pkern** to get the names of kernel primitives associated functions, the so-called *eXecutionTokens* "XT", for stack manipulation or other functions:
  - *k_dup()* to duplicate TOS
  - *k_swap()* to exchange TOS with NOS
  - ...

```
---
ex> getcwd: """          # define a word to get the current working directory
""> import os
""> print(os.getcwd())
""> """
Compile warning: all data after closing triple-quote in current line is ignored!
```

```
co> ; .
ex> cwd: getcwd exec ; .
 Data stack empty
ex> cwd .
c:\main\Python
 Data stack empty
ex>
```

  An useful word to list the content of a given directory:

```
ex> getdir: """
""> import os
""> dpush(os.listdir(path=tos()))
""> """
Compile warning: all data after closing triple-quote in current line is ignored!
co> ; .
 Data stack empty
ex> dir: getdir exec plist ; .
 Data stack empty
ex> 'c:/main/Python/advancedrpn dir .
   0 calculator.py
   1 main1.py
   2 pycalc.py
   3 settings.json
   4 __init1__.py
   5 __main__.py
   6 __pycache__
 Data stack items: 1
['c:/main/Python/advancedrpn']
ex>
```

  It's up to you to add *testing of valid input*...otherwise you
risk to be *aborted by Python with a traceback* and *loose* all your
work (if you were not smart enough to save it before experimenting...):

```
ex> 'c:/main/Python/zzzzzzz dir .
Traceback (most recent call last):
  File "c:\main\Python\rppy.py", line 4318, in <module>
    NEXT()
  File "c:\main\Python\rppy.py", line 430, in NEXT
    exec(ExecList[IP][CFA])
  File "<string>", line 1, in <module>
  File "c:\main\Python\rppy.py", line 4043, in k_exec
    exec(dpop())
  File "<string>", line 3, in <module>
FileNotFoundError: [WinError 3] The system cannot find the path specified:
'c:/main/Python/zzzzzzz'
```

  And all your data is lost ...
  To avoid such mishap, use the well known *pair try-except*:

```
---
ex> getdir: """
""> import os
""> try:
"">     dpush(os.listdir(path=tos()))
""> except(AttributeError,TypeError,OSError) as e:
"">     abort(str(e))
""> """
Compile warning: all data after closing triple-quote in current line is ignored!
co> ; .
 Data stack empty
ex> dir: getdir exec plist ; .
 Data stack empty
ex> 'c:/main/Python/zzzzzzz dir .
"exec" aborted: [WinError 3] The system cannot find the path specified:
'c:/main/Python/zzzzzzz'
 Aborted at Execution List Index:  7
 In definition: "dir" at index: 5
  TOS:  <class 'str'> c:/main/Python/zzzzzzz
 Return stack items:1
1 return from: dir
 Data stack items: 1
['c:/main/Python/zzzzzzz']
ex>
---
```

Now the error is catched *inside RPPy*, continue working in all tranquility...

And a last example, to print today's date:

```
---
ex> getdate: """
""> import datetime as dt
""> dpush(dt.date.today())
""> """
Compile warning: all data after closing triple-quote in current line is ignored!
co> ; .
 Data stack empty
ex> date: getdate exec print ; .
 Data stack empty
ex> date .
2024-11-17
 Data stack empty
ex>
---
```

Error Handling

Generally speaking, there are three kinds of errors encountered:

- *compile errors*:
  - unknown word (typo error or unwanted space amid space-free lists, strings,etc.)
  - definition doesn't start at input line beginning
  - unmatched pairs if-then
  - absence of closing markers for strings
  - wrong syntax in expressions (strings without blanks as input to evaluate by RPPy's Read-Eval-Print-Loop or separate **eval** word)

  What's important to know: compiling is done all time around, not only at defining a new word; RPPy *always generates compiled code*, which, if not used in a definition, is discarded after execution. So don't be surprised if it says "compile error xxx" even if the input prompt is "ex> " and not "co> "

- *runtime errors*:
  - mostly these are sort of:
    - missing arguments
    - wrong argument type
    - wrong order of arguments; if you are lucky, the bad placed argument generates a type error; if not, you could get a semantic error

- *semantic and logical errors*:
  - no error is signalled, but the output is not what was expected.

  Hard to debug, try avoiding them by thoroughly testing every new definition before it's included in the next ones. Include a data stack printing **pds** at word start and before return, where you guess it may be a problem; later on you can remove it by redefining that word. Or, if redefining is not wanted, use vectored execution inserting an indirect call to **pds**, removed finally if it's all ok (see "Control Flow – Vectored Execution"). But generally, as said multiple times before, define *short words*, a small couple of lines long. It's much easier to make them work as expected, compared to a maze of imbricated comparisons, loops and branches in a n-lines huge definition, not even mentioning all the stackrobatics involved.

Now, specific for RPPy as opposed to Python, an error doesn't abruptly end your program with a traceback; instead you continue to write corrected data/words from where on the error was detected, no matter if inside or outside a definition. Let's see concrete examples:

```
ex> 20 45 swap - 33 xx 44 * .
Compile error: name 'xx' is not defined in expression: xx
 Undefined: "xx"
ex>                        # all part before xx is executed
 Data stack items: 2     # 20 45 swap - is 25 , then 33 pushed
[25, 33]                   # but all part after xx is not executed
ex>
```

  Continue from where the error appeared:

```
ex> + 44 * .
 Data stack items: 1
[2552]
ex>
```


```
ex> 22 33 add 99 print .
"add" aborted: First argument for "add" must be a set
 Aborted at Execution List Index:  33
 Aborted in execution string or in definition with multiple returns
  TOS:  <class 'int'> 33
  NOS:  <class 'int'> 22
 Data stack items: 2
[22, 33]
ex>
```

  Same as before: **22** and **33** pushed on the stack, but **99 print** ignored, therefore get correct arguments for add, then do the rest.
  As a rule of thumb, get assured that the stack contains *valid data* before continuing input after an error point.

  Correcting errors inside definitions obeys the same rule; you simply continue with the corrected part:

```
ex> baddef: 11 22 swax 33 + ; .
Compile error: name 'swax' is not defined in expression: swax
 Undefined: "swax"
co> swap 33 + ; .       # RPPy remains in compile mode, so you can
 Data stack empty       # continue with corrected data
ex> 'baddef pdef .
  baddef      at index:  2 ""  ""
  baddef: 11 22 swap 33 + ; # corrected def

  No duplicate (older definitions) present
 Data stack empty
ex>
```

  NB. There's no need to start again from the beginning with the definition, continue it as said before.

  A runtime error example:

```
ex> zdiv: 0 / ; .
 Data stack empty
ex> aa: 99 zdiv ; .
 Data stack empty
ex> aa .
"/" aborted: division by zero
```

```
 Aborted at Execution List Index:  16
 In definition: "zdiv" at index: 14
  TOS:  <class 'int'> 0
  NOS:  <class 'int'> 99
 Return stack items:1
1 return from: aa
 Data stack items: 2
[99, 0]
ex> 14 pexlst .          # see where the error appears
   14 zdiv:
   15 lit  0             # wrong data in zdiv
   16 /
   17 ;
   18 aa:
   19 lit  99
   20 jump zdiv
   21 call aa
   22 lit  14
   23 pexlst
 Data stack items: 2
[99, 0]
ex>
```

Here execution of your words stops and control is returned to the Read-Eval-Print-Loop of RPPy; unfortunately there's no traceback, only the faulty word appears in the abort message .

A last word about unexpected errors during RPPy execution *exiting with a Python traceback*: the kernel primitives include all sort of exception handling internally, but that’s not exhaustive. Various combination can appear leading to errors not thought about. Please send a copy of the traceback and of the saved definitions (if available) to maintainer *raberpy4@outlook.com* And don’t forget to **save** often your set of definitions...

Glossary of RPPy kernel primitives

###1	===Data Stack words===
dup	(n -- n n) duplicate TOS
2dup	(x y -- x y x y) duplicate NOS and TOS
drop	(n --) drop TOS
2drop	(x y --) drop TOS and NOS
swap	(x y -- y x) exchange TOS with NOS
over	(x y -- x y x) copy NOS over TOS
nip	(x y -- y) drop NOS
tuck	(x y -- y x y) insert TOS before NOS
rot	(x y z -- y z x) rotate rightwise top three items
-rot	(x y z -- z x y) rotate leftwise top three items
pick	(xn xn-1 xn-2 ... x0 n -- xn xn-1 xn-2 ... x0 xn) replace TOS with a copy of the n-th item
cleards	(... --) empty data stack
###2	===User Stack words===
uspush	(n --) data stack | (-- n) user stack ; push TOS to user stack
uspop	(-- n) data stack | (n --) user stack ; pop TOS of user stack & push to data stack
clearus	(... --) empty user stack
###3	===Math words===
+	(x y -- x+y) add TOS to NOS
-	(x y -- x-y) subtract TOS from NOS
*****	(x y -- x*y) multiply TOS by NOS
/	(x y -- x/y) divide NOS by TOS - floating point div
//	(x y -- x//y) divide NOS by TOS - floored (integer) div
%	(x y -- rem) remainder of x/y division
divmod	(x y -- quot rem) quotient & remainder of x/y division
**	(x y -- x**y) NOS at power of TOS
++	(n -- n+1) increment TOS by 1
--	(n -- n-1) decrement TOS by 1
neg	(n -- neg(n)) negate n
abs	(x -- abs(x)) absolute value of x
round	(n i -- round(n)) n rounded to i digits
min	(seq -- min(seq)) minimum of values in sequence seq
max	(seq -- max(seq)) maximum of values in sequence seq
sum	(seq -- sum(seq)) sum of items in sequence seq
val+	(n1 n2 ... ni i -- sum(ni)) sum of top i values, i >= 1
val*	(n1 n2 ... ni i -- product(ni)) product of top i values, i >= 1
str+	(str1 str2 ... stri i -- str) concatenate top i strings, i >= 1
lst+	(list1 list2 ... listi i -- list) concatenate top i lists, i >=1

###4 ===Comparison words===

<	(x y -- x y) ZF = 1 if x<y
>	(x y -- x y) ZF = 1 if x>y
<=	(x y -- x y) ZF = 1 if x<=y
>=	(x y -- x y) ZF = 1 if x>=y
==	(x y -- x y) ZF = 1 if x==y
!=	(x y -- x y) ZF = 1 if x!=y
<0	(n -- n) ZF = 1 if n < 0
>0	(n -- n) ZF = 1 if n > 0
=0	(n -- n) ZF = 1 if n == 0
!=0	(n -- n) ZF = 1 if n != 0
<0>	(idx1 idx2 idx3 n --) if n<0 execute word with idx1; if n=0 execute idx2; else idx3
<=>	(idx1 idx2 idx3 x y --) if x<y execute word with idx1; if x=y execute idx2, else idx3
zf=	(n --) pop TOS to ZF
zf	(-- ZF) push ZF
in	(seq item -- seq item) ZF = 1 if item found in sequence
notin	(seq item -- seq item) ZF = 1 if item not found in sequence
is	(item1 item2 -- item1 item2) ZF = 1 if item1 and item2 are the same object
isnot	(item1 item2 -- item1 item2) ZF = 1 if item1 and item2 are not the same object

###5 ===Logical words===

and	(x y -- x y) ZF = 1 if both x and y are True
or	(x y -- x y) ZF = 1 if either x or y or both are True
xor	(x y -- x y) ZF = 1 if either (x is True and y is False) or (x is False and y is True)
not	(n -- n) ZF = 1 if n is False or None
-not	(n -- n) ZF = 1 if n is True

###6 ===Bitwise words===

&	(x y -- x&y) bitwise and
|	(x y -- x|y) bitwise or
^	(x y -- x^y) bitwise exclusive or
~	(n -- ~n) one's complement
<<	(n i -- n) shift left n by i bits
>>	(n i -- n) shift right n by i bits

###7 ===Assignment words===

=	(n idx --) store n to var[idx]: variable with index idx
+=	(n idx --) var[idx] += n
-=	(n idx --) var[idx] -= n
***=**	(n idx --) var[idx] *= n
/=	(n idx --) var[idx] /= n
//=	(n idx --) var[idx] //= n
%=	(n idx --) var[idx] %= n
****=**	(n idx --) var[idx] **= n

###8 ===Index Register words===

i=	(n --) pop TOS to I
i	(-- I) push I
i++	(--) I += 1
i--	(--) I -= 1
j=	(n --) pop TOS to J
j	(-- J) push J
j++	(--) J += 1
j--	(--) J -= 1
k=	(n --) pop TOS to K
k	(-- K) push K
k++	(--) K += 1
k--	(--) K -= 1
iloop	(idx --) execute word with index idx I times
jloop	(idx --) execute word with index idx J times
kloop	(idx --) execute word with index idx K times

###9 ===Printer words===

pds	(--) print data stack
pdson	(--) switch data stack print to ON
pdsoff	(--) switch data stack print to OFF
pus	(--) print user stack
prs	(--) print return stack
pdefall	(--) print all definitions dictionary
pdef	(defname --) print a definition
pkernall	(--) print all kernel primitives dictionary
pkern	(--) print kernel primitives per category
pexlst	(idx --) print execution list starting with index idx
plist	(list --) print list as enumerated items
pcomp	(--) print compiler definitions dictionary
print	(item --) print item
printf	(formatstr n --) print n according to format string
printnl	(--) print newline
printend	(item endstr --) print item followed by end string

###10 ===List words===

list (str -- list) create list from str
listcre ("[]" item1 item2 ... itemn -- list) create list of items
listexp (list -- item1 item2 ... itemn) expand list of items
append (list item -- list) append item to list
clear (list -- []) clear list
copy (list -- list listcopy) return original and shallow copy of list
count (str substr -- n) return count of substr occurences in str
extend (list iterable -- list) extend list with iterable
index (list item -- index) return index of item in list
insert (list index item -- list) insert item before index in list
len (list -- n) return lenght of list
pop (list -- list item) pop last item from list
popidx (list index -- list item) pop item at index in list
remove (list index -- list) remove item at index in list
reverse (list -- list) reverse list in place
sortasc (list -- list) sort list in ascending order
sortdes (list -- list) sort list in descending order
s[i] (list -- list item) extract item at index given in register I
s[i:j] (list -- list slice) extract slice given in registers I:J
s[i:] (list -- list slice) extract slice starting at I until end
s[:j] (list -- list slice) extract slice from start until J
s[i][j] (list -- list item) extract item at J from item at I
s[i][j][k] (list -- list item) extract item at K from item at J from item at I
del[i] (list -- list) delete item at index given in register I
del[i:j] (list -- list) delete slice given in registers I:J
del[i:] (list -- list) delete slice starting at I until end
del[:j] (list -- list) delete slice from start until J
del[i][j] (list -- list) delete item at J from item at I
del[i][j][k] (list -- list) delete item at K from item at J from item at I

###11 ===Dictionary words===

dict (str -- dict) create dictionary from str
dictcre (keylist valuelist -- dict) create dictionary from list of keys & values
dictexp (dict -- keylist valuelist) expand dictionary to list of keys & values
get (dict key -- value) get value for key
items (dict -- list) return a list of all dict items as tuples (key,value)
popkey (dict key -- dict value) return and remove value for key
popitem (dict -- dict item) return and remove last item (Python 3.7+)
setdefault (dict key value -- dict) insert the pair key:value
update (dict seq -- dict) update dict with sequence seq

###12 ===Tuple words===

tuple (str -- tuple) create tuple from str
tupcre (valuelist -- tuple) create tuple from list of values
tupexp (tuple -- valuelist) expand tuple to list of values

###13 ===Set words===

setstr (str -- set) create set from single string
set (stritems -- set) create set from string of multiple items
setcre (valuelist -- set) create set from list of values
setexp (set -- valuelist) expand set to list of values
add (set item -- set) add item to set
difference (set1 set2 -- set) return set of differences between set1 and set2
sdifference (set1 set2 -- set) return set of symmetric differences between set1 and set2
udifference (set1 set2 -- set) differences update: remove all elements of set2 from set1
discard (set item -- set) remove item from set
intersection (set1 set2 -- set) return intersection of set1 and set2
isdisjoint (set1 set2 --) ZF=1 if intersection of set1/set2 is null
issubset (set1 set2 --) ZF=1 if set1 is a subset of set2
issuperset (set1 set2 --) ZF=1 if set1 is a superset of set2
union (set1 set2 -- set) return union of set1 with set2

###14 ===String words===

capitalize (str -- str) return a string with first char capitalized, rest is lowercase
casefold (str -- str) return a string with all chars lowercase
center (str width -- str) return centered string of lenght width, space padded
centerfill (str width fill -- str) return centered string of lenght width, fill char padded
emptystr (-- "") leave an empty string
endswith (str suffix --) ZF=1 if str ends with suffix
expandtabs (str tabsize -- str) return string with tabs expanded
find (str substr -- n) return lowest index of substr found in str, -1 if not found
isalnum (str --) ZF=1 if all chars in str are alphanumeric
isalpha (str --) ZF=1 if all chars in str are alphabetic
isascii (str --) ZF=1 if all chars in str are ASCII or str is empty
isdecimal (str --) ZF=1 if all chars in str are decimal
isdigit (str --) ZF=1 if all chars in str are digits
islower (str --) ZF=1 if all chars in str are lowercase
isnumeric (str --) ZF=1 if all chars in str are numeric
isprintable (str --) ZF=1 if all chars in str are printable or str is empty
isspace (str --) ZF=1 if all chars in str are whitespaces
istitle (str --) ZF=1 if str is titlecased
isupper (str --) ZF=1 if all chars in str are uppercase
join (str seq -- str) return a string concatenated with all the strings in seq
ljust (str width -- str) return left justified string of width lenght, space padded
ljustfill (str width fill -- str) return left justified string of width lenght, fill char padded
lower (str -- str) return a string with all chars converted to lowercase
lstrip (str -- str) return a string with all leading spaces removed
lstripchr (str strchr -- str) return a string with all leading strchr chars combinations removed

partition (str strsep -- tuple) split str at first occurence of strsep; return a 3-element tuple: part before separator, separator itself, part after separator
removeprefix (str strpref -- str) return a string with strpref removed
removesuffix (str strsuff -- str) return a string with strsuff removed
replace (str oldstr newstr -- str) return a string with all occurences of oldstr replaced by newstr
replacecnt (str oldstr newstr count -- str) return a string with count occurences of oldstr replaced by newstr
rfind (str substr -- n) return highest index of substr found in str, -1 if not found
rjust (str width -- str) return right justified string of width lenght, space padded
rjustfill (str width fill -- str) return right justified string of width lenght, fill char padded
rpartition (str strsep -- tuple) split str at last occurence of strsep; return a 3-element tuple: part before separator, separator itself, part after separator
rsplit (str strsep maxsplit -- list) return list of maxsplit words splitted from right by strsep
rstrip (str -- str) return a string with all trailing spaces removed
rstripchr (str strchr -- str) return a string with all trailing strchr chars combinations removed
split (str strsep maxsplit -- list) return list of maxsplit words splitted by strsep
splitlines (str -- list) return list of lines split at line breaks without including line breaks
splitlnbrk (str -- list) return list of lines split at line breaks, including line breaks
startswith (str prefix --) ZF=1 if str starts with prefix
strip (str -- str) return a string with all spaces removed
stripchr (str strchr -- str) return a string with all strchr chars combinations removed
swapcase (str -- str) return a string with uppercase chars converted to lovercase and vice versa
title (str -- str) return a titlecased string with words starting with uppercase
upper (str -- str) return a string with all chars converted to uppercase
zfill (str width -- str) return a string of lenght width left filled with "0"

###15 ===File I/O words===
open (filename filemode -- filehandle) open filename with given filemode
read (filehandle -- filecontent) read whole file
readline (filehandle -- linecontent) read a single line from file
readlines (filehandle -- list) read all lines into list
readsize (filehandle size -- sizecontent) read size bytes from file
seek (filehandle offset origin -- n) file content pointer n = origin+offset
tell (filehandle -- n) n = current pointer in file
write (filehandle str -- n) write str to file, n= nb. of chars written
close (filehandle --) close file

###16 ===JSON words===

jsdump (filename pyobj --) write Python object as JSON object to filename
jsload (filename -- pyobj) load JSON object as Python object from filename
jsdumps (pyobj -- jsonstring) code Python object to JSON string
jsloads (jsonstring -- pyobj) decode JSON string to Python object

###17 ===Edit/Load/Save words===

deldef (defname --) delete definition defname
edit (defname --) edit definition defname
refdef (defname -- list) list of all definitions including a reference to defname
repdef (defname --) replace old definitions of defname with the last one defined
load (filename --) load RPPy user generated words from filename
save (filename --) save RPPy user generated words to filename
s. (--) save RPPy user generated words to file "tempsave.rpp"

###18 ===Miscellaneous words===

abort (str --) print abort message str, switch to REPL
all (seq --) ZF=1 if all elements in seq are true or seq is empty
any (seq --) ZF=1 if any element in seq is true; ZF=0 if seq is empty
bin (n -- strbin) convert integer n to binary string
chr (n -- strchr) convert integer n to string representing associated glyph
complex (str -- n) convert str to complex number n
choose (idx1 idx2 --) if ZF=1 execute word with idx1, else idx2
enumerate (seq -- list) list of tuples (count,value) iterating over seq
eval (str -- item) TOS = eval(str)
execidx (idx --) execute word with index idx
exec (str -- ...) exec(str), stack depends of what exec does
float (str/int -- n) convert string or integer to floating point number
format (formatstr n -- str) convert value n to str according to formatstr
help (--) print help chapters
hex (n -- strhex) convert integer n to hexadecimal string
input (-- str) read a line from input and convert data to a string
inputprompt (strprompt -- str) write strprompt to standard output, then read a line
int (str/n -- n) convert str or number n to integer n
intbase (str base -- n) convert str to integer n according to base
intro (--) print introduction to RPPy
license (--) print RPPy license
oct (n -- stroct) convert integer n to octal string
ord (strchr -- n) convert string representing one character to integer n
quit (... -- ...) end execution of RPPy
str (item -- str) return a string interpretation of item
type (item -- itemtype) TOS = type of item

Glossary of compiler words

===Compiler words===

#	(--) comment, skip rest of line
.	(--) mark end of compile phase, start execution phase
""	(--) mark start of docstring to embed in definition
"""	(--) mark start of multiline string
"	(--) mark start of string with blanks and without double quotes
->	(--) mark start of string with blanks/quotes of all sort
;	(--) compile return or compile jump (if tail call optimisation)
if	(--) compile if: continue execution if ZF == 1, else branch to "then"
ifz	(--) compile ifz: same as if, but for ZF == 0
ifneq	(x y -- x y) compile ifneq: continue execution if x!=y, else branch to "then"
then	(--) compile then: branch there if condition not satisfied

Index of RPPy kernel primitives in alphabetical order

0 (!= , [k_neq() , (x y -- x y) ZF = 1 if x!=y])
1 (!=0 , [k_zneq() , (n -- n) ZF = 1 if n != 0])
- indexes of word category 2..19 eliminated (===...===)
20 (% , [k_rem() , (x y -- rem) remainder of x/y division])
21 (%= , [k_remstore() , (n idx --) var[idx] %= n])
22 (& , [k_bitand() , (x y -- x&y) bitwise and])
23 (* , [k_star() , (x y -- x*y) multiply TOS by NOS])
24 (** , [k_dblstar() , (x y -- x**y) NOS at power of TOS])
25 (**= , [k_dblstarstore() , (n idx --) var[idx] **= n])
26 (*= , [k_starstore() , (n idx --) var[idx] *= n])
27 (+ , [k_plus() , (x y -- x+y) add TOS to NOS])
28 (++ , [k_plusone() , (n -- n+1) increment TOS by 1])
29 (+= , [k_plusstore() , (n idx --) var[idx] += n])
30 (- , [k_minus() , (x y -- x-y) subtract TOS from NOS])
31 (-- , [k_minusone() , (n -- n-1) decrement TOS by 1])
32 (-= , [k_minusstore() , (n idx --) var[idx] -= n])
33 (-not , [k_notnot() , (n -- n) ZF = 1 if n is True])
34 (-rot , [k_rotl() , (x y z -- z x y) rotate leftwise top three items])
35 (/ , [k_slash() , (x y -- x/y) divide NOS by TOS - floating point div])
36 (// , [k_dblslash() , (x y -- x//y) divide NOS by TOS - floored (integer) div])
37 (//= , [k_dblslashstore() , (n idx --) var[idx] //= n])
38 (/= , [k_slashstore() , (n idx --) var[idx] /= n])
39 (2drop , [k_ddrop() , (x y --) drop TOS and NOS])
40 (2dup , [k_ddup() , (x y -- x y x y) duplicate NOS and TOS])
41 (< , [k_le() , (x y -- x y) ZF = 1 if x<y])
42 (<0 , [k_zle() , (n -- n) ZF = 1 if n < 0])
43 (<0> , [k_lesszeq() , (idx1 idx2 idx3 n --) if n<0 execute word with idx1; if n=0 execute idx2; else idx3])
44 (<< , [k_lshift() , (n i -- n) shift left n by i bits])
45 (<= , [k_leeq() , (x y -- x y) ZF = 1 if x<=y])
46 (<=> , [k_lesseqgt() , (idx1 idx2 idx3 x y --) if x<y execute word with idx1; if x=y execute idx2, else idx3])
47 (= , [k_store() , (n idx --) store n to var[idx]: variable with index idx])
48 (=0 , [k_zeq() , (n -- n) ZF = 1 if n == 0])
49 (== , [k_eq() , (x y -- x y) ZF = 1 if x==y])
50 (> , [k_gt() , (x y -- x y) ZF = 1 if x>y])
51 (>0 , [k_zgt() , (n -- n) ZF = 1 if n > 0])
52 (>= , [k_gteq() , (x y -- x y) ZF = 1 if x>=y])
53 (>> , [k_rshift() , (n i -- n) shift right n by i bits])
54 (^ , [k_bitxor() , (x y -- x^y) bitwise exclusive or])
55 (abort , [k_abort() , (str --) print abort message str, switch to REPL])
56 (abs , [k_abs() , (x -- abs(x)) absolute value of x])
57 (add , [k_add() , (set item -- set) add item to set])
58 (all , [k_all() , (seq --) ZF=1 if all elements in seq are true or seq is empty])
59 (and , [k_and() , (x y -- x y) ZF = 1 if both x and y are True])

60 (any , [k_any() , (seq --) ZF=1 if any element in seq is true; ZF=0 if seq is empty])
61 (append , [k_append() , (list item -- list) append item to list])
62 (bin , [k_bin() , (n -- strbin) convert integer n to binary string])
63 (capitalize , [k_capitalize() , (str -- str) return a string with first char capitalized, rest is lowercase])
64 (casefold , [k_casefold() , (str -- str) return a string with all chars lowercase])
65 (center , [k_center() , (str width -- str) return centered string of lenght width, space padded])
66 (centerfill , [k_centerfill() , (str width fill -- str) return centered string of lenght width, fill char padded])
67 (choose , [k_choose() , (idx1 idx2 --) if ZF=1 execute word with idx1, else idx2])
68 (chr , [k_chr() , (n -- strchr) convert integer n to string representing associated glyph])
69 (clear , [k_clear() , (list -- []) clear list])
70 (cleards , [k_clrdstk() , (... --) empty data stack])
71 (clearus , [k_clrustk() , (... --) empty user stack])
72 (close , [k_close() , (filehandle --) close file])
73 (complex , [k_complex() , (str -- n) convert str to complex number n])
74 (copy , [k_copy() , (list -- list listcopy) return original and shallow copy of list])
75 (count , [k_count() , (str substr -- n) return count of substr occurences in str])
76 (del[:j] , [k_deljstart() , (list -- list) delete slice from start until J])
77 (del[i:] , [k_deliend() , (list -- list) delete slice starting at I until end])
78 (del[i:j] , [k_delij() , (list -- list) delete slice given in registers I:J])
79 (del[i] , [k_deli() , (list -- list) delete item at index given in register I])
80 (del[i][j] , [k_delijsecond() , (list -- list) delete item at J from item at I])
81 (del[i][j][k] ,[k_delijkthird() , (list -- list) delete item at K from item at J from item at I])
82 (deldef , [k_deldef() , (defname --) delete definition defname])
83 (dict , [k_dict() , (str -- dict) create dictionary from str])
84 (dictcre , [k_dictcre() , (keylist valuelist -- dict) create dictionary from list of keys & values])
85 (dictexp , [k_dictexp() , (dict -- keylist valuelist) expand dictionary to list of keys & values])
86 (difference , [k_difference() , (set1 set2 -- set) return set of differences between set1 and set2])
87 (discard , [k_discard() , (set item -- set) remove item from set])
88 (divmod , [k_divmod() , (x y -- quot rem) quotient & remainder of x/y division])
89 (drop , [k_drop() , (n --) drop TOS])
90 (dup , [k_dup() , (n -- n n) duplicate TOS])
91 (edit , [k_edit() , (defname --) edit definition defname])
92 (emptystr , [k_emptystr() , (-- "") leave an empty string])
93 (endswith , [k_endswith() , (str suffix --) ZF=1 if str ends with suffix])
94 (enumerate , [k_enumerate() , (seq -- list) list of tuples (count,value) iterating over seq])
95 (eval , [k_evaluate() , (str -- item) TOS = eval(str)])

96 (exec , [k_exec() , (str -- ...) exec(str), stack depends of what exec does])
97 (execidx , [k_execidx() , (idx --) execute word with index idx])
98 (expandtabs , [k_expandtabs() , (str tabsize -- str) return string with tabs expanded])
99 (extend , [k_extend() , (list iterable -- list) extend list with iterable])
100 (find , [k_find() , (str substr -- n) return lowest index of substr found in str, -1 if not found])
101 (float , [k_float() , (str/int -- n) convert string or integer to floating point number])
102 (format , [k_format() , (formatstr n -- str) convert value n to str according to formatstr])
103 (get , [k_get() , (dict key -- value) get value for key])
104 (help , [h.k_help() , (--) print help chapters])
105 (hex , [k_hex() , (n -- strhex) convert integer n to hexadecimal string])
106 (i , [k_I() , (-- I) push I])
107 (i++ , [k_incI() , (--) I += 1])
108 (i-- , [k_decI() , (--) I -= 1])
109 (i= , [k_setI() , (n --) pop TOS to I])
110 (iloop , [k_Iloop() , (idx --) execute word with index idx I times])
111 (in , [k_in() , (seq item -- seq item) ZF = 1 if item found in sequence])
112 (index , [k_index() , (list item -- index) return index of item in list])
113 (input , [k_input() , (-- str) read a line from input and convert data to a string])
114 (inputprompt , [k_inputprompt() , (strprompt -- str) write strprompt to standard output, then read a line])
115 (insert , [k_insert() , (list index item -- list) insert item before index in list])
116 (int , [k_int() , (str/n -- n) convert str or number n to integer n])
117 (intbase , [k_intbase() , (str base -- n) convert str to integer n according to base])
118 (intersection , [k_intersection() , (set1 set2 -- set) return intersection of set1 and set2])
119 (intro , [h.k_intro() , (--) print introduction to RPPy])
120 (is , [k_is() , (item1 item2 -- item1 item2) ZF = 1 if item1 and item2 are the same object])
121 (isalnum , [k_isalnum() , (str --) ZF=1 if all chars in str are alphanumeric])
122 (isalpha , [k_isalpha() , (str --) ZF=1 if all chars in str are alphabetic])
123 (isascii , [k_isascii() , (str --) ZF=1 if all chars in str are ASCII or str is empty])
124 (isdecimal , [k_isdecimal() , (str --) ZF=1 if all chars in str are decimal])
125 (isdigit , [k_isdigit() , (str --) ZF=1 if all chars in str are digits])
126 (isdisjoint , [k_isdisjoint() , (set1 set2 --) ZF=1 if intersection of set1/set2 is null])
127 (islower , [k_islower() , (str --) ZF=1 if all chars in str are lowercase])
128 (isnot , [k_isnot() , (item1 item2 -- item1 item2) ZF = 1 if item1 and item2 are not the same object])

```
129 ( isnumeric ,[ k_isnumeric() ,  ( str -- ) ZF=1 if all chars in str are numeric ])
130 ( isprintable , [ k_isprintable() ,  ( str -- ) ZF=1 if all chars in str are printable
                    or str is empty ])
131 ( isspace ,  [ k_isspace() ,  ( str -- ) ZF=1 if all chars in str are whitespaces ])
132 ( issubset , [ k_issubset() ,  ( set1 set2 -- ) ZF=1 if set1 is a subset of set2 ])
133 ( issuperset , [ k_issuperset() ,  ( set1 set2 -- ) ZF=1 if set1 is a superset of
                    set2 ])
134 ( istitle ,  [ k_istitle() ,  ( str -- ) ZF=1 if str is titlecased ])
135 ( isupper ,  [ k_isupper() ,  ( str -- ) ZF=1 if all chars in str are uppercase ])
136 ( items ,  [ k_items() ,  ( dict -- list ) return a list of all dict items as tuples
                    (key,value) ])
137 ( j ,  [ k_J() ,  ( -- J ) push J ])
138 ( j++ ,  [ k_incJ() ,  ( -- ) J += 1 ])
139 ( j-- ,  [ k_decJ() ,  ( -- ) J -= 1 ])
140 ( j= ,  [ k_setJ() ,  ( n -- ) pop TOS to J ])
141 ( jloop ,  [ k_Jloop() ,  ( idx -- ) execute word with index idx J times ])
142 ( join ,  [ k_join() ,  ( str seq -- str ) return a string concatenated with all
                    the strings in seq ])
143 ( jsdump ,  [ k_jsdump() ,  ( filename pyobj -- ) write Python object as JSON
                    object to filename ])
144 ( jsdumps , [ k_jsdumps() ,  ( pyobj -- jsonstring ) code Python object to JSON
                    string ])
145 ( jsload ,  [ k_jsload() ,  ( filename -- pyobj ) load JSON object as Python
                    object from filename ])
146 ( jsloads ,  [ k_jsloads() ,  ( jsonstring -- pyobj ) decode JSON string to Python
                    object ])
147 ( k ,  [ k_K() ,  ( -- K ) push K ])
148 ( k++ ,  [ k_incK() ,  ( -- ) K += 1 ])
149 ( k-- ,  [ k_decK() ,  ( -- ) K -= 1 ])
150 ( k= ,  [ k_setK() ,  ( n -- ) pop TOS to K ])
151 ( kloop ,  [ k_Kloop() ,  ( idx -- ) execute word with index idx K times ])
152 ( len ,  [ k_len() ,  ( list -- n ) return lenght of list ])
153 ( license ,  [ k_license() ,  ( -- ) print RPPy license ])
154 ( list ,  [ k_list() ,  ( str -- list ) create list from str  ])
155 ( listcre ,  [ k_listcre() ,  ( "[]" item1 item2 ... itemn -- list ) create list of
                    items ])
156 ( listexp ,  [ k_listexp() ,  ( list -- item1 item2 ... itemn ) expand list of
                    items ])
157 ( ljust ,  [ k_ljust() ,  ( str width -- str ) return left justified string of width
                    lenght, space padded ])
158 ( ljustfill ,  [ k_ljustfill() ,  ( str width fill -- str ) return left justified string of
                    width lenght, fill char padded ])
159 ( load ,  [ k_load() ,  ( filename -- ) load RPPy user generated words from
                    filename ])
160 ( lower ,  [ k_lower() ,  ( str -- str ) return a string with all chars converted
                    to lowercase ])
161 ( lst+ ,  [ k_sumlst() ,  ( list1 list2 ... listi i -- list ) concatenate top i  lists,
                    i >=1 ])
162 ( lstrip ,  [ k_lstrip() ,  ( str -- str ) return a string with all leading spaces
                    removed ])
163 ( lstripchr ,  [ k_lstripchr() ,  ( str strchr -- str ) return a string with all leading
```

```
                strchr chars combinations removed ])
164 ( max ,     [ k_max() ,  ( seq -- max(seq) ) maximum of values in sequence
                seq ])
165 ( min ,     [ k_min() ,  ( seq -- min(seq) ) minimum of values in sequence seq
                ])
166 ( neg ,     [ k_negate() ,  ( n -- neg(n) ) negate n ])
167 ( nip ,     [ k_nip() ,  ( x y -- y ) drop NOS ])
168 ( not ,     [ k_not() ,  ( n -- n ) ZF = 1 if n is False or None ])
169 ( notin ,   [ k_notin() ,  ( seq item -- seq item ) ZF = 1 if item not found in
                sequence ])
170 ( oct ,     [ k_oct() ,  ( n -- stroct ) convert integer n to octal string ])
171 ( open ,    [ k_open() ,  ( filename filemode -- filehandle ) open filename with
                given filemode ])
172 ( or ,      [ k_or() ,  ( x y -- x y ) ZF = 1 if either x or y or both are True ])
173 ( ord ,     [ k_ord() ,  ( strchr -- n ) convert string representing one character
                to integer n ])
174 ( over ,    [ k_over() ,  ( x y -- x y x ) copy NOS over TOS ])
175 ( partition , [ k_partition() ,  ( str strsep -- tuple ) split str at first occurence of
                strsep; return a 3-element tuple: part before separator, separator
                itself, part after separator ])
176 ( pcomp ,   [ k_pcompdef() ,  ( -- ) print compiler definitions dictionary ])
177 ( pdef ,    [ k_pddef() ,  ( defname -- ) print a definition ])
178 ( pdefall , [ k_pdefall() ,  ( -- ) print all definitions dictionary ])
179 ( pds ,     [ k_pds() ,  ( -- ) print data stack ])
180 ( pdsoff ,  [ k_pdsoff() ,  ( -- ) switch data stack print to OFF ])
181 ( pdson ,   [ k_pdson() ,  ( -- ) switch data stack print to ON ])
182 ( pexlst ,  [ k_pexlst() ,  ( idx -- ) print execution list starting with index
                idx ])
183 ( pick ,    [ k_pick() ,  ( xn xn-1 xn-2 ... x0 n -- xn xn-1 xn-2 ... x0 xn )
                replace TOS with a copy of the n-th item ])
184 ( pkern ,   [ k_pkerndef() ,  ( -- ) print kernel primitives per category ])
185 ( pkernall , [ k_pkerndefall() ,  ( -- ) print all kernel primitives dictionary ])
186 ( plist ,   [ k_plist() ,  ( list -- ) print list as enumerated items ])
187 ( pop ,     [ k_pop() ,  ( list -- list item ) pop last item from list ])
188 ( popidx ,  [ k_popidx() ,  ( list index -- list item ) pop item at index in list ])
189 ( popitem , [ k_popitem() ,  ( dict -- dict item ) return and remove last item
                (Python 3.7+) ])
190 ( popkey ,  [ k_popkey() ,  ( dict key -- dict value ) return and remove value
                for key ])
191 ( print ,   [ k_print() ,  ( item -- ) print item ])
192 ( printend , [ k_printend() ,  ( item endstr -- ) print item followed by end string
                ])
193 ( printf ,  [ k_printf() ,  ( formatstr n -- ) print n according to format string ])
194 ( printnl ,  [ k_printnl() ,  ( -- ) print newline ])
195 ( prs ,     [ k_prs() ,  ( -- ) print return stack ])
196 ( pus ,     [ k_pus() ,  ( -- ) print user stack ])
197 ( quit ,          [ k_quit() ,  ( ... -- ... ) end execution of RPPy ])
198 ( read ,    [ k_read() ,  ( filehandle -- filecontent ) read whole file ])
199 ( readline , [ k_readline() ,  ( filehandle -- linecontent ) read a single line from
                file ])
200 ( readlines ,[ k_readlines() ,  ( filehandle -- list ) read all lines into list ])
```

201 (readsize , [k_readsize() , (filehandle size -- sizecontent) read size bytes from file])
202 (refdef , [k_refdef() , (defname -- list) list of all definitions including a reference to defname])
203 (remove , [k_remove() , (list index -- list) remove item at index in list])
204 (removeprefix , [k_removeprefix() , (str strpref -- str) return a string with strpref removed])
205 (removesuffix , [k_removesuffix() , (str strsuff -- str) return a string with strsuff removed])
206 (repdef , [k_repdef() , (defname --) replace old definitions of defname with the last one defined])
207 (replace , [k_replace() , (str oldstr newstr -- str) return a string with all occurences of oldstr replaced by newstr])
208 (replacecnt , [k_replacecnt() , (str oldstr newstr count -- str) return a string with count occurences of oldstr replaced by newstr])
209 (reverse , [k_reverse() , (list -- list) reverse list in place])
210 (rfind , [k_rfind() , (str substr -- n) return highest index of substr found in str, -1 if not found])
211 (rjust , [k_rjust() , (str width -- str) return right justified string of width lenght, space padded])
212 (rjustfill , [k_rjustfill() , (str width fill -- str) return right justified string of width lenght, fill char padded])
213 (rot , [k_rot() , (x y z -- y z x) rotate rightwise top three items])
214 (round , [k_round() , (n i -- round(n)) n rounded to i digits])
215 (rpartition ,[k_rpartition() , (str strsep -- tuple) split str at last occurence of strsep; return a 3-element tuple: part before separator, separator itself, part after separator])
216 (rsplit , [k_rsplit() , (str strsep maxsplit -- list) return list of maxsplit words splitted from right by strsep])
217 (rstrip , [k_rstrip() , (str -- str) return a string with all trailing spaces removed])
218 (rstripchr , [k_rstripchr() , (str strchr -- str) return a string with all trailing strchr chars combinations removed])
219 (s. , [k_sdot() , (--) save RPPy user generated words to file "tempsave.rpp"])
220 (s[:j] , [k_sjstart() , (list -- list slice) extract slice from start until J])
221 (s[i:] , [k_siend() , (list -- list slice) extract slice starting at I until end])
222 (s[i:j] , [k_sij() , (list -- list slice) extract slice given in registers I:J])
223 (s[i] , [k_si() , (list -- list item) extract item at index given in register I])
224 (s[i][j] , [k_sijsecond() , (list -- list item) extract item at J from item at I])
225 (s[i][j][k] , [k_sijkthird() , (list -- list item) extract item at K from item at J from item at I])
226 (save , [k_save() , (filename --) save RPPy user generated words to filename])
227 (sdifference , [k_sdifference() , (set1 set2 -- set) return set of symmetric differences between set1 and set2])
228 (seek , [k_seek() , (filehandle offset origin -- n) file content pointer n = origin+offset])

```
229 ( set ,         [ k_set() ,  ( stritems -- set ) create set from string of multiple
                    items ])
230 ( setcre ,      [ k_setcre() ,  ( valuelist -- set ) create set from list of values ])
231 ( setdefault , [ k_setdefault() ,  ( dict key value -- dict ) insert the pair
                    key:value ])
232 ( setexp ,     [ k_setexp() ,  ( set -- valuelist ) expand set to list of values ])
233 ( setstr ,     [ k_setstr() ,  ( str -- set ) create set from single string ])
234 ( sortasc ,    [ k_sortasc() ,  ( list -- list ) sort list in ascending order ])
235 ( sortdes ,    [ k_sortdes() ,  ( list -- list ) sort list in descending order ])
236 ( split ,      [ k_split() ,  ( str strsep maxsplit -- list ) return list of maxsplit
                    words splitted by strsep ])
237 ( splitlines , [ k_splitlines() ,  ( str -- list ) return list of lines split at line breaks
                    without including line breaks ])
238 ( splitlnbrk ,      [ k_splitlnbrk() ,  ( str -- list ) return list of lines split at line
                    breaks, including line breaks ])
239 ( startswith ,[ k_startswith() ,  ( str prefix -- ) ZF=1 if str starts with prefix ])
240 ( str ,        [ k_str() ,  ( item -- str ) return a string interpretation of item ])
241 ( str+ ,       [ k_sumstr() ,  ( str1 str2 ... stri i -- str ) concatenate top i strings,
                    i >= 1 ])
242 ( strip ,      [ k_strip() ,  ( str -- str ) return a string with all spaces removed ])
243 ( stripchr ,   [ k_stripchr() ,  ( str strchr -- str ) return a string with all strchr
                    chars combinations removed ])
244 ( sum ,        [ k_sum() ,  ( seq -- sum(seq) ) sum of  items in sequence seq ])
245 ( swap ,       [ k_swap() ,  ( x y -- y x ) exchange TOS with NOS ])
246 ( swapcase ,[ k_swapcase() ,  ( str -- str ) return a string with uppercase chars
                    converted to lovercase and vice versa ])
247 ( tell ,       [ k_tell() ,  ( filehandle -- n ) n = current pointer in file ])
248 ( title ,      [ k_title() ,  ( str -- str ) return a titlecased string with words
                    starting with uppercase ])
249 ( tuck ,       [ k_tuck() ,  ( x y -- y x y ) insert TOS before NOS ])
250 ( tupcre ,     [ k_tupcre() ,  ( valuelist -- tuple ) create tuple from list of
                    values ])
251 ( tupexp ,     [ k_tupexp() ,  ( tuple -- valuelist ) expand tuple to list of values ])
252 ( tuple ,      [ k_tuple() ,  ( str -- tuple ) create tuple from str  ])
253 ( type ,       [ k_type() ,  ( item -- itemtype ) TOS = type of item ])
254 ( udifference , [ k_udifference() ,  ( set1 set2 -- set ) differences update:
                    remove all elements of set2 from set1 ])
255 ( union ,      [ k_union() ,  ( set1 set2 -- set ) return union of set1 with set2 ])
256 ( update ,     [ k_update() ,  ( dict seq -- dict ) update dict with sequence seq ])
257 ( upper ,      [ k_upper() ,  ( str -- str ) return a string with all chars converted
                    to uppercase ])
258 ( uspop ,      [ k_uspop() ,  ( -- n ) data stack | ( n -- ) user stack ; pop TOS of
                    user stack & push to data stack ])
259 ( uspush ,     [ k_uspush() ,  ( n -- ) data stack | ( -- n ) user stack ; push TOS
                    to user stack ])
260 ( val* ,       [ k_mulval() ,  ( n1 n2 ... ni i -- product(ni) ) product of top i
                    values, i >= 1 ])
261 ( val+ ,       [ k_sumval() ,  ( n1 n2 ... ni i -- sum(ni) ) sum of top i values, i >=
                    1 ])
262 ( write ,      [ k_write() ,  ( filehandle str -- n ) write str to file, n= nb. of chars
                    written ])
```

```
263 ( xor ,      [ k_xor() ,  ( x y -- x y ) ZF = 1 if either (x is True and y is False)
                 or (x is False and y is True) ])
264 ( zf ,       [ k_ZF() ,  ( -- ZF ) push ZF ])
265 ( zf= ,      [ k_setZF() ,  ( n -- ) pop TOS to ZF ])
266 ( zfill ,    [ k_zfill() ,  ( str width -- str ) return a string of lenght width left
                 filled with "0"  ])
267 ( | ,        [ k_bitor() ,  ( x y -- x|y ) bitwise or ])
268 ( ~ ,        [ k_bitcompl() , "( n -- ~n ) one s complement"])
```

===///===