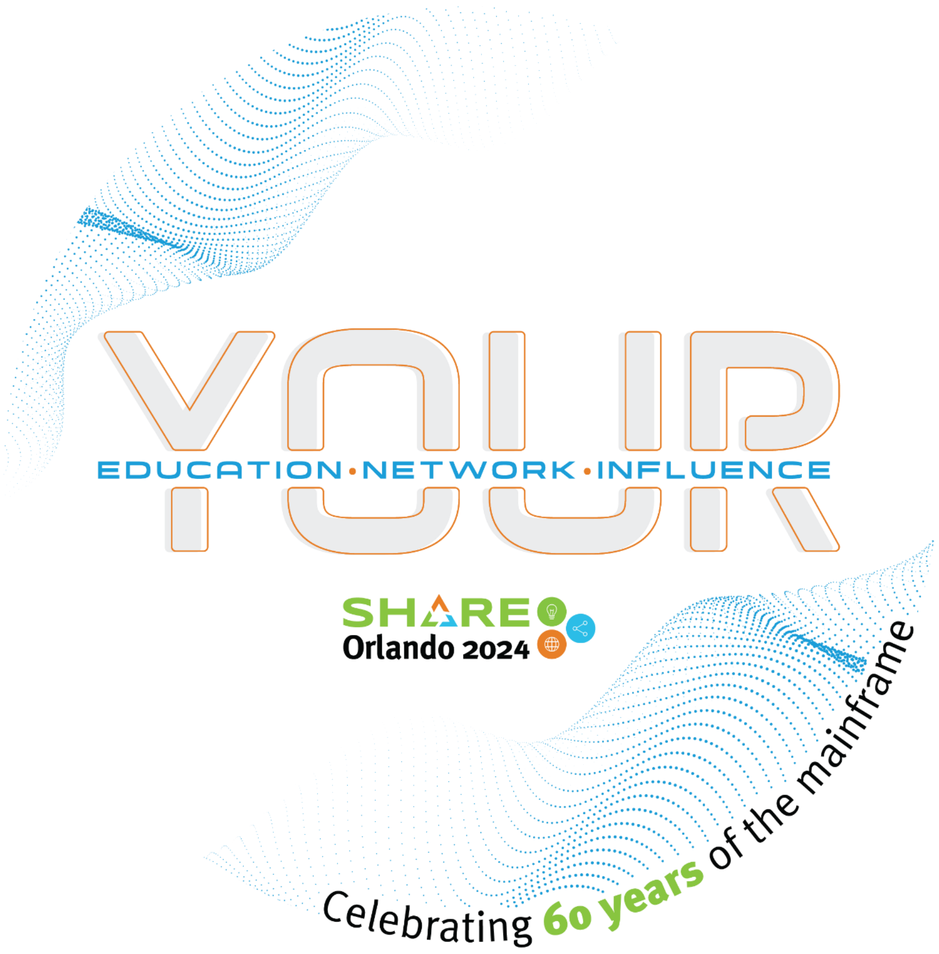


YOUR
EDUCATION•NETWORK•INFLUENCE
SHARE
Orlando 2024
Celebrating 60 years of the mainframe

Python For The Rexx Programmer

(and maybe the other way, too)

Session 52622

Ray Mullins, Broadcom MSD

Based on an original presentation by David Crayford, Rocket Software

Presentation Notes

- This presentation is originally written by David Crayford of Rocket Software.
- The subject came up in a conversation on the System Z Enthusiasts Discord Server, as we both had the same idea.
- He graciously offered to share the presentation with me, as he likely will not be able to attend SHARE in the near future.
- I owe him several beers for this.
- The original presentation included live demos. I did not have time to integrate neither live nor recorded demos for this first iteration. (I hope to at the next SHARE.) I was also not able to run under USS, so there are some output examples missing.

Agenda

- IBM Open Enterprise SDK for Python overview
- Language constructs
- Code dive

IBM OPEN ENTERPRISE SDK FOR PYTHON OVERVIEW

IBM Open Enterprise SDK for Python overview

- Native Python compiler and interpreter for z/OS
- z/OS UNIX only using enhanced ASCII (ISO 8859-1)
- Python standard library. The library includes highly used programming tasks in areas like string operations, cryptology, threading, networking, internet and web service tools, operating system interfaces, and protocols.
- zIIP eligible (up to 70%)

Why Python?

- World’s most popular programming language. Huge talent pool.
- Easy to learn
- Massive standard library and eco-system
- Well suited to data science and analytics
- Powerful language used in many different industries:
  - ChatGPT
  - Tesla auto-pilot
  - Instagram
- Strategic on z/OS
  - Big push for next gen tooling based on Ansible and other automation platforms.

Why Python?

- New and mid-career mainframe staff
  - Sharing samples
  - Wanting to use it in place of Rexx
    - OPS/MVS customers are interested in using Python in rules (OPS/REXX ADDRESS USS)

LANGUAGE CONSTRUCTS

Language Constructs

	Rexx	Python
Paradigm	Procedural (imperative)*	Multi-paradigm, object-oriented, procedural (imperative), functional, reflective
Designed by	Mike Cowlishaw	Guido van Rossum
Year first released onto an unsuspecting public	1979	1991
Standard	ANSI X3.274 (1996); *The Rexx Language, 2nd Edition*	Python Software Foundation
Typing	Dynamic	Duck, dynamic, strong, optional type annotations
Reserved words	No	Yes
Guiding Influence	The Principle of Least Astonishment	Monty Python’s Flying Circus

*-Only Rexx for z/OS, z/VM, and VSEn is discussed here. Object-oriented Rexx versions have not been ported to those platforms.

Language Constructs

	Rexx	Python
Keywords	Case insensitive	Case sensitive
Identifiers	Case insensitive	Case sensitive
Comments	/* line comment */ /* multi-line comments */	# line comment ''' multi-line comments '''
Structure blocks	DO/END, SELECT/END, subroutines/procedures/functions*	Indentation (white space count), functions

*-Function packages are beyond the scope of this presentation.

Operators–Arithmetic

	REXX	Python	Note
Addition	+	+	
Subtraction	-	-	
Multiplication	*	*	
Division	/	/	
Floor division	%	//	
Remainder/Modulo	//	%	REXX is not modulo, because the result may be negative. In Python the result is always positive.
Power	**	**	REXX only supports whole numbers so you can’t calculate a square root using `n ** .5`

Operators–Assignment

	REXX	Python	Note
Direct Assignment	=	=	a = 1970
Addition assignment	N/A	+=	n += 1
Subtraction assignment	N/A	-=	n -= 41
Multiplication assignment	N/A	*=	n *= 17
Division assignment	N/A	/=	n /= 32
Exponent assignment	N/A	**=	n **= 10

Operators–Comparison

	REXX	Python	Note
Equal to	=	==	
Not equal to	\=, ¬=, /=, <>, ><	!=	Rexx has three ways to say "not", as well as "if it’s not less than or greater than, it must be not equal!"
Greater than	>	>	
Less than	<	<	
Greater than or equal to	>=, \<, ¬<	>=	Rexx allows you to say "not less than"
Less than or equal to	<=, \>, ¬>	<=	Rexx allows you to say "not greater than"

Operators–Comparison (cont'd)

	REXX	Python	Note
Strictly equal (identical)	==	N/A	The strict operators do not strip blanks before comparing strings. Also, 0 == 0.1 tests false but true when using =
Strictly not equal (inverse of ==)	\\==, ¬==, /==	N/A	
Strictly greater than	>>	N/A	
Strictly less than	<<	N/A	
Strictly greater than or equal	>>=	N/A	
Strictly not less than	\\<<, ¬<<	N/A	
Strictly less than or equal	<<=	N/A	
Strictly not greater than	\\>>, ¬>>	N/A	

Operators–Logical

	REXX	Python	Note
Logical AND	&	and	
Logical OR	|	or	
Exclusive OR (returns true if either bit, but not both, is true)	&&	N/A	Python employs the XOR bitwise operator to implement the equivalent functionality.
Logical NOT	\\,¬	not	

Operators–Bitwise

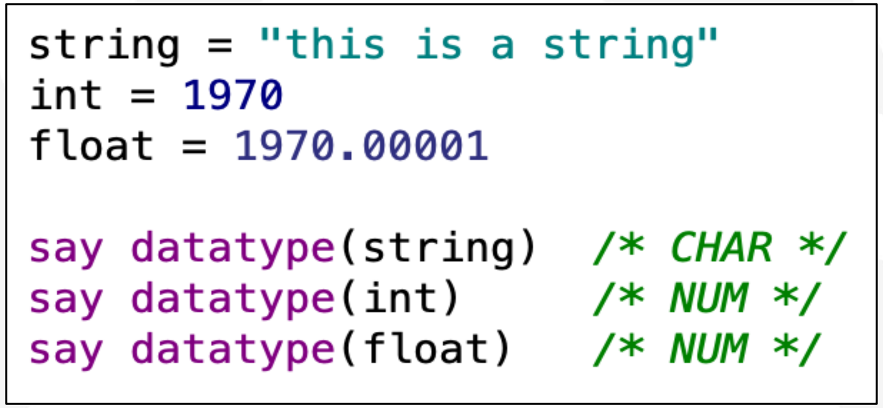
	REXX	Python	Note
Bitwise AND	BITAND()	&	
Bitwise OR	BITOR()	|	

Bitwise XOR	BITXOR()	^	This operator can be used to emulate REXX XOR.
			`a = True; b = False`
			`print(a ^ b) # True`

Bitwise NOT	N/A	~	
Bitwise right shift	N/A	>>	
Bitwise left shift	N/A	<<	

Types–Rexx

REXX only has one data type, which is the string. Conversions between strings and numeric types are achieved through coercion.

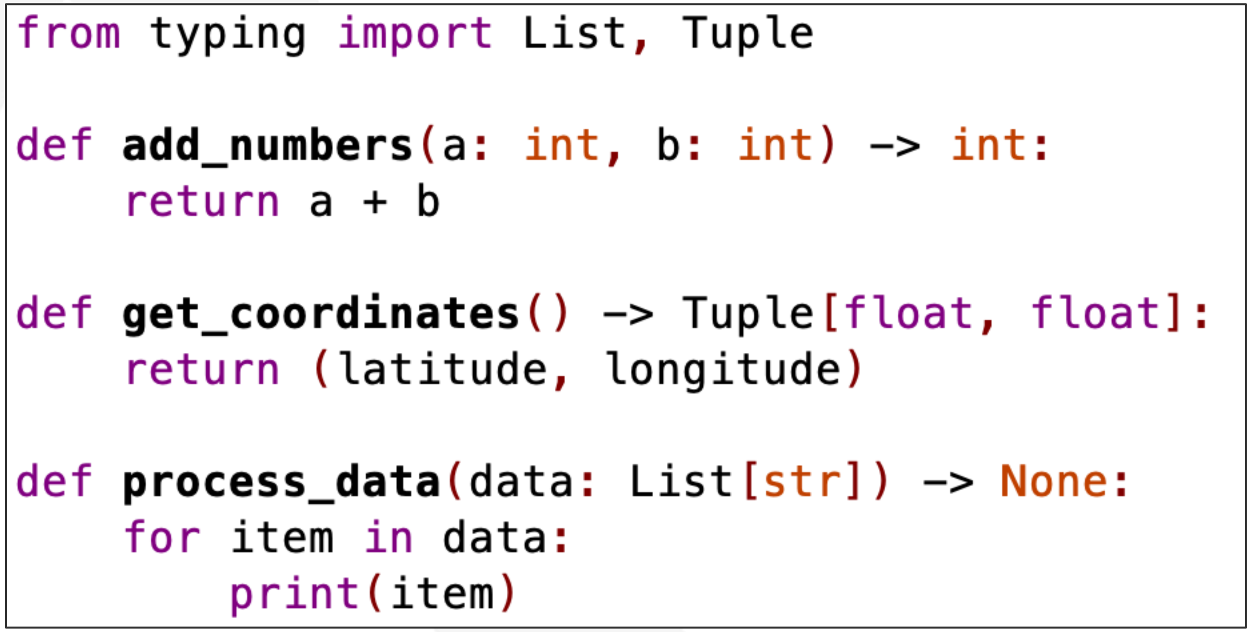


```
string = "this is a string"
int = 1970
float = 1970.00001

say datatype(string)  /* CHAR */
say datatype(int)     /* NUM */
say datatype(float)   /* NUM */
```

Types–Python

Python provides a comprehensive set of built-in data types, each tailored for specific purposes in programming. These built-in data types provide the building blocks for constructing complex data structures and solving a wide range of computational problems.



```
from typing import List, Tuple

def add_numbers(a: int, b: int) -> int:
    return a + b

def get_coordinates() -> Tuple[float, float]:
    return (latitude, longitude)

def process_data(data: List[str]) -> None:
    for item in data:
        print(item)
```

Types–Python

Python type hinting is a feature that allows you to annotate variables, function parameters, and return values with hints about their expected data types. These hints improve code readability and help catch potential type-related errors during development. They are not enforced at runtime but assist static analysis tools and IDEs.

Text	`str`
Numeric	`int, float, complex`
Sequence	`list, tuple, range`
Mapping	`dict`
Set	`set, frozenset`
Boolean	`bool`
Binary	`bytes, bytearray, memoryview`
NoneType	`None`

Questions?

CODE DIVE

Structure

```
/* REXX */

main:
  call hello

  exit 0  /* must be explicit or falls through */

hello:
  say "hello"
  return
```

NO!

```
hello()

def hello():  # functions are first class objects!!!
    print("hello")
```

YES!

```
def main():
    hello()

def hello():
    print("hello")

if __name__ == '__main__':
    main()   # Note: Python uses (significant) whitespace to delimit blocks
```

Conditionals

```
/* REXX */
mood = "happy"

if mood = "happy" then do
  say "I'm very pleased your're in good mood"
end
else if mood = "sad" then do
  say "Cheer up!"
end
else if mood = "anxious" | mood = "worried" then do
    say "Take a chill pill bro!"
end
else do
  say "I'll put the kettle on!"
end

select
  when mood = "happy" then do
    say "I'm very pleased your're in good mood"
  end
  when mood = "sad" then do
    say "Cheer up"
  end
  when mood = "anxious" | mood = "worried" then do
    say "Take a chill pill bro!"
  end
  otherwise do
    say "I'll put the kettle on!"
  end
 end
```

```
mood = "anxious"

if mood == "happy":
    print("I'm very pleased your're in good mood")
elif mood == "sad":
    print("Cheer up!")
elif mood in ("anxious", "worried"):
    print("Take a chill pill bro!")
else:
    print("I'll put the kettle on!")

match (mood):
    case "happy":
        print("I'm very pleased you're in a good mood")
    case "sad":
        print("Cheer up")
    case "anxious" | "worried":
        print("Take a chill pill bro!")
    case _:
        print("I'll put the kettle on!")
```

Loops–For

```
/* REXX */

say " "
say "-- REXX loops --"

/* controlled repetitive loop */
do i = 1 to 10
  say I
end

do i = 3 to -2 by -1
  say i
end

/* conditional phases (while and until)*/
do i = 1 to 10 by 2 until i > 6
  say I
end

say ""
do i = 1 to 10 by 2 while i < 6
  say I
end
```

```
print("\n-- python loops --")

for i in range(1, 11):
    print(i)

for i in range(3, -3, -1):
    print(i)

for i in range(1, 11, 2):
    print(i)
    if i > 6:
        break

for i in range(1, 11, 2):
    if i > 6:
        break
    print(i)
```

Python output

```
-- python loops --
1
2
3
4
5
6
7
8
9
10
3
2
1
0
-1
-2
1
3
5
7
1
3
5
```

Loops–While/Until

```
/* REXX */

stuff_to_do = 1

do while stuff_to_do
  nop
end

finished = 0

do until finished
  nop
end

do forever
  if is_true() then iterate
  else leave
end

is_true:
  return random(0, 1)
```

```
import random

stuff_to_do = True

while stuff_to_do:
    pass
else:
    print("There was nothing to do")

finished = False

while not finished:
    pass

def is_true(): return random.choice([True, False])

while True:
    if is_true():
        continue
    else:
        break
```

Functions

```
/* REXX */

call print "hello", "world" /* parenthesis are optional */

say "results of add = " add(1, 2, 3, 4, 5, 6)

exit 0

echo:
  parse arg first_word, second_word
  say first_word second_word
  return

/* variable number of arguments */

add:
  res = 0
  do i = 1 to arg()
    res = res + arg(i)
  end
  return res
```

```
def echo(first_word: str, second_word: str):
    print(first_word, second_word)

# variable number of arguments (tuple)
def add(*numbers):
    total = 0
    for num in numbers:
        total += num
    return total

echo("hello", "world")
echo(second_word="le monde", first_word="bonjour")

print(f"result of add = {add(1,2,3,4,5,6)}")

# keyword arguments
def total_fruits(**kwargs):
    print(kwargs, type(kwargs))

total_fruits(banana=5, mango=7, apple=8)

# output:
#
# {'banana': 5, 'mango': 7, 'apple': 8} <class 'dict'>
```

Python output

```
hello world
bonjour le monde
result of add = 21
{'banana': 5, 'mango': 7, 'apple': 8} <class 'dict'>
```

Functions–Scope

```
/* REXX */

signal on novalue

x = 50
call first
exit

first: procedure
    call second
    return

second: procedure expose x
    say "X-squared is" x ** x
    return

third:
    side_effects = "yep"
    return

novalue:
  say 'NOVALUE raised at line' sigl
  say 'The referenced variable is' "CONDITION"('D’)
  exit 8

glbl._screen_height = 25
glbl._screen_width = 80
glbl._attributes = 31

process_screen: expose glbl.
    say glbl._screen_height
    return
```

```
x = 50

def first():
    x = 20  # local variable
    second()

def first_mutate():
    global x  # declare global x so we can change it
    x = 20
    second()

def second():
    print(f"X-squared is {x**x}")

first()
first_mutate()
```

Python output

```
X-squared is
88817841970012523233890533447265625000000000000000000000000000000
00000000000000000000

X-squared is 104857600000000000000000000
```

Functions–Closures

```
def counter(max: int):
    current = 0

    def get_next():
        nonlocal current
        if current == max:
            return None
        current += 1
        return current

    return get_next

my_counter = counter(10)
while (num := my_counter()):  # Note: Walrus operator (3.8)
    print(num)
```

Python output

```
1
2
3
4
5
6
7
8
9
10
```

Functions–Generators

```
def counter(max: int):
    current = 0
    while current <= max:
        yield current  # suspends and yields to caller
        current += 1

for num in counter(10):
    print(num)

# generator expression (also called a generator comprehension)
numbers = (num for num in counter(10))
```

Python output

```
0
1
2
3
4
5
6
7
8
9
10
```

Functions–Decorators

```
from datetime import datetime, timezone

# logging decorator
def logger(fn):
   def inner(*args, **kwargs):
        called_at = datetime.now(timezone.utc)
        to_execute = fn(*args, **kwargs)
        print(f'{fn.__name__} executed. Logged at {called_at}’)
        return to_execute
    return inner

@logger
def fee():
    pass

@logger
def fo():
    pass

@logger
def fum():
    pass

fee()
fo()
fum()
```

Python output

```
fee executed. Logged at 2024-03-03 19:49:59.325279+00:00
fo executed. Logged at 2024-03-03 19:49:59.326280+00:00
fum executed. Logged at 2024-03-03 19:49:59.326280+00:00
```

Strings

```
/* REXX */

scale = "Do Re Mi Fa Sol La Ti Do"

/* quick and dirty ;-) */
do i = 1 to words(scale)
  say word(scale, i)
end

/* the right way :-) */
line = scale
do while line \= ‘’
  parse var line w line
  say w
end

/* count words in a string */
say "They're are "words(scale)" notes in the major scale"

/* print the characters in the string */
len = length(scale)
do i = 1 to len
  say substr(scale, i, 1)
end

/* concatenate */
scale = "  "scale "  "
say "'"scale "'"

/* trim */
scale = strip(scale, 'L')
scale = strip(scale, 'T')
scale = strip(scale)  /* strip(scale, 'B') */
```

```
import re

scale = "Do Re Mi Fa Sol La Ti Do"

for w in scale.split():
    print(w)

# count words in a string
print(f"They're are {len(scale.split())} notes in the major scale")

# print the characters in the string
for char in scale:
    print(char)

# concatenate
scale = "  " + scale + "  "
print(f"'{scale}'")

# trim
scale = scale.lstrip()
scale = scale.rstrip()
scale = scale.strip()
```

Strings (cont.)

```
/* sub-strings */
s = left(scale, 2)
s = right(scale, 2)
s = substr(scale, 3, 2)

/* with justification */
str = "REXX"
say left(str, length(str), 10) /* 'REXX      ' */
```

```
# sub-strings using slicing
s = scale[:2]   # left(scale, 2)
s = scale[:-2]  # right(scale, 2)
s = scale[2:4]  # substr(scale, 3, 2)

""" str[start:stop:step] # start through not past stop, by step
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
   0   1   2   3   4   5
  -6  -5  -4  -3  -2  -1
"""

# with justification
print("Python".ljust(10))  # 'Python    '

# wordpos
phrase = "Mi"
words = scale.split()
pos = words.index(phrase) if phrase in words else None
print(f"wordpos={pos}")  # pos relative from zero

# wordindex
def wordindex(string, pos):
    n = 0
    for match in re.finditer(r'\S+', scale):
        n += 1
        if n == pos:
            return match.start() + 1 # <-- not pythonic!
    return None

print(f'wordindex={wordindex(scale, 2)}')
```

Python output

```
Do
Re
Mi
Fa
Sol
La
Ti
Do
They're are 8 notes in the major scale
D
o

R
e

M
i

F
a

S
o
l

L
a

T
i

D
o
'  Do Re Mi Fa Sol La Ti Do  '
Python
wordpos=2
wordindex=4
```

Parse

```
/* REXX */

dsname = "MY.DATASET.NAME(@README)"

parse var dsname dsn '(' member ‘)’

say "dsn="dsn" member="member

/* a more vexing parse that doesn't work as expected!
*/
parm = "DSNAME(MY.DATASET.NAME(@README))"

parse var parm 'DSNAME(' dsn ‘)’
say dsn  /* MY.DATASET.NAME(@README */

parse var parm 'DSNAME(' dsn '(' mem')' ')’
say dsn mem /* MY.DATASET.NAME @README */

parm = "DSNAME(MY.DATASET.NAME)"
parse var parm 'DSNAME(' dsn '(' mem ')' ')’
say dsn mem /* MY.DATASET.NAME)  */
```

```
import re
from parse import parse, compile

dsname = "MY.DATASET.NAME(@README)"

r = re.compile(r'(\S+)\((.*?)\)')
result = r.search(dsname).groups()
print(f"dsn={result[0]} member={result[1]}")

# example of using an anchor to match end of string
parm = "DSNAME(MY.DATASET.NAME(@README))"

r = re.compile(r'DSNAME\((.*?)\)$')
result = r.search(parm).groups()
print(result[0])

# Use the 'parse' library to approximate template parsing
dsn, member = parse("{}({})", dsname)
print(f"dsn={dsn} member={member}")

# use a precompiled parser
p = compile("{}({})")
dsn, member = p.parse(dsname)
print(f"dsn={dsn} member={member}")
```

Parse (cont.)

```
/* sub-strings */
s = left(scale, 2)
s = right(scale, 2)
s = substr(scale, 3, 2)

/* with justification */
str = "REXX"
say left(str, length(str), 10) /* 'REXX      ’ */
```

```
# sub-strings using slicing
s = scale[:2]   # left(scale, 2)
s = scale[:-2]  # right(scale, 2)
s = scale[2:4]  # substr(scale, 3, 2)

""" str[start:stop:step] # start through not past stop, by step
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
   0   1   2   3   4   5
  -6  -5  -4  -3  -2  -1
"""

# with justification
print("Python".ljust(10))  # 'Python    '

# wordpos
phrase = "Mi"
words = scale.split()
pos = words.index(phrase) if phrase in words else None
print(f"wordpos={pos}")  # pos relative from zero

# wordindex
def wordindex(string, pos):
    n = 0
    for match in re.finditer(r'\S+', scale):
        n += 1
        if n == pos:
            return match.start() + 1 # <-- not pythonic!
    return None

print(f'wordindex={wordindex(scale, 2)}')
```

Miss Rexx function names? forbiddenfruit!

```
from forbiddenfruit import curse, reverse
from tnz import rexx

# Hmmm, that's not very cool!
print(rexx.wordpos("Club", "Brentford Football Club"))

# Let's be subversive and monkey patch the Python standard library string Type ＼(＾O＾)／
curses = {'left': rexx.left, 'right': rexx.right, 'substr': rexx.substr, 'subword': rexx.subword}

for name, function in curses.items():
    curse(str, name, function)

team = "Chelsea FC"
print("'" + team.left(30) + "’")
print(team.subword(2))

for name in curses.keys():
    reverse(str, name)
```

Data structures–Lists

```
/* REXX */

text.1 = "this is a text of record"
text.2 = "to demonstrate how to process"
text.3 = "texts in REXX"
text.0 = 3


n = text.0 + 1
text.0 = n
text.n = "oh yeah!"

do i = 1 to text.0
  say text.i
end
```

```
text = [
    "this is a list of records",
    "to demonstrate how to process",
    "lists in Python"
]

text.append("oh yeah!")

for rec in text:
    print(rec)

# this is a list of records to demonstrate how to process lists in Python oh yeah!
print(' '.join(text))

text.sort()
text.sort(reverse=True)

# Use a list comprehenstion to create a list of squared numbers
squares = [ n*n for n in range(0, 10)]
```

Python output

```
this is a list of records
to demonstrate how to process
lists in Python
oh yeah!
this is a list of records to demonstrate how to process lists in Python oh yeah!
```

Data structures–Tuples

```
cars = ('BMW', 'Tesla', 'Ford', 'Toyota’)
print('Total Items:', len(cars))
for car in cars:
    print(car)

# convert tuple to a list
car_list = list(cars)
print("Converting tuple to list:", car_list)
print("Type", type(car_list))
```

Python output

```
Total Items: 4
BMW
Tesla
Ford
Toyota
Converting tuple to list: ['BMW', 'Tesla', 'Ford', 'Toyota']
Type <class 'list'>
```

Data structures–Dictionaries (basic)

```
/* REXX */

null = '00’x

dict. = null

call dict_add "color", "red"
call dict_add "make", "Toyota"

say "color="dict_get("color")
say dict_get("novalue")

call dict_delete "color"

call dict_clear

say "make="dict_get("make")

exit

dict_add: procedure expose dict.
  parse arg key, value
  dict.key = value
  return

dict_get: procedure expose dict.
  parse arg key
  return dict.key

dict_delete: procedure expose dict.
  parse arg key
  dict.key = null   /* <<< BUG!! */
  return

dict_clear: procedure expose dict.
  drop dict.
  dict. = '00’x
  return
```

```
kvdict = {}

kvdict["color"] = "red"

print(f"color={kvdict['color']}")

try:
    print(f"color={kvdict['NoValue']}")
except KeyError as ex:
    print(f"Dictionary key was not found: {ex}")

value: str = kvdict.get("NoValue")
assert value == None

if 'NoValue' in kvdict:  # check first to prevent KeyError exceptions
    del kvdict['NoValue']

# more pythonic then 'del'
kvdict.pop('color', "default")

kvdict = {'color': 'red', 'make': 'Toyota', 'model': 'Camry'}

for key in kvdict.keys(): # for key in kvdict:
    print(key)

for k, v in kvdict.items():
    print(f'{k}={v}')
```

Python output

```
color=red
Dictionary key was not found: 'NoValue'
color
make
model
color=red
make=Toyota
model=Camry
```

Data structures–Dictionaries (complex)

```
/* REXX */

system.server.type = "MVS"
system.server.name = "SYS6"
system.server.plex = "REPLEX"
system.server.jobs.0 = 4

system.server.jobs.1.jobname = "BACKUP"
system.server.jobs.1.jobno = "J00012"
system.server.jobs.2.jobname = "PAYROLL"
system.server.jobs.2.jobno = "J00013"
system.server.jobs.3.jobname = "BILLING"
system.server.jobs.3.jobno = "J00014"
system.server.jobs.4.jobname = "ORDERS"
system.server.jobs.4.jobno = "J00015"

say "System name . . . :" system.server.name
say "System type . . . :" system.server.type
do i = 1 to system.server.jobs.0
  say "jobname="system.server.jobs.i.jobname ||,
      " jobno="system.server.jobs.i.jobno
end
```

```
import inspect


system = {
    "server": {
        "name": "SYS6",
        "type": "MVS",
        "plex": "REPLEX",
        "jobs": [
            {"jobname": "BACKUP", "jobno": "J00012"},
            {"jobname": "PAYROLL", "jobno": "J00013"},
            {"jobname": "BILLING", "jobno": "J00014"},
            {"jobname": "ORDERS", "jobno": "J00015"},
        ]
    }
}

print(f"System name . . . : {system['server']['name']}")
print(f"System type . . . : {system['server']['type']}")
for job in system['server']['jobs']:
    print(f"jobname={job['jobname']} jobno={job['jobname']}")

print(system['server'])
```

Python output

```
System name . . . : SYS6
System type . . . : MVS
jobname=BACKUP jobno=BACKUP
jobname=PAYROLL jobno=PAYROLL
jobname=BILLING jobno=BILLING
jobname=ORDERS jobno=ORDERS
{'name': 'SYS6', 'type': 'MVS', 'plex': 'REPLEX', 'jobs': [{'jobname': 'BACKUP',
'jobno': 'J00012'}, {'jobname': 'PAYROLL', 'jobno': 'J00013'}, {'jobname': 'BILLING',
'jobno': 'J00014'}, {'jobname': 'ORDERS', 'jobno': 'J00015'}]}
```

Data structures–Stacks & Queues

```
/* REXX */

push '1. last’
push '2. in’
push '3. last’
push '4. out’

do queued()
  parse pull item
  say item
end

queue '1. first’
queue '2. in’
queue '3. first’
queue '4. out’

push '0. <== yay ==>’


do queued()
  parse pull item
  say item
end
```

```
from collections import deque

stack = []

stack.append('1. last')
stack.append('2. in')
stack.append('3. last')
stack.append('4. out')

while len(stack) > 0:
    print(stack.pop())

queue = deque()

queue.append('1. first')
queue.append('2. in')
queue.append('3. first')
queue.append('4. out')

queue.appendleft('0. <== yay ==>')

while len(queue):
    print(queue.popleft())
```

Python output

```
4. out
3. last
2. in
1. last
0. <== yay ==>
1. first
2. in
3. first
4. out
```

Data structures–Sets

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

print(x.intersection(y))
print(x.union(y))
print(x.difference(y))

if "apple" in x:
    print("success!")
```

Python output

```
{'apple'}
{'microsoft', 'banana', 'cherry', 'apple', 'google'}
{'cherry', 'banana'}
success!
```

File I/O (from a z/OS perspective)

```
/* REXX */

"ALLOC FI(INPUT) DA('SYSTEM.DAILY.SMF')
SHR"

 do until eof
    "EXECIO 10000 DISKR INPUT ( STEM rec."
    eof = (rc > 0)
    do i = 1 to rec.0
      iterate
    end
  end

"EXECIO 0 DISKR INPUT ( FINIS"

"FREE FI(INPUT)"
```

```
from pyzfile import *

# read a binary file using QSAM
try:
    with ZFile("//'SYSTEM.DAILY.SMF'", "rb,type=record,noseek") as file:
        for rec in file:
            pass
except ZFileError as e:
    print(e)

# read a text file using BSAM
try:
    with ZFile("//'USERID.CNTL(JCL)'", "rb,type=record",encoding='cp1047') as
file:
        for rec in file:
            print(rec)
except ZFileError as e:
    print(e)

# Note: pyzfile can handle all types of data set: BSAM, QSAM, VSAM (ESDS, KSDS,
RRDS), Unix
```

Control block access

```
/* REXX - CPU ids */

cvt      = storage(10,4)
cvtsname = storage(d2x(x2d(c2x(cvt))+x2d(154)),8)
ecvt     = storage(d2x(c2d(cvt)+x2d(8c)),4)
ipa      = storage(d2x(c2d(ecvt)+x2d(188)),4)
ipasxnam = storage(d2x(x2d(c2x(ipa))+x2d(160)),8)
say "SYSNAME="||strip(cvtsname)
"SYSPLEX="||strip(ipasxnam)  "Date="||date()

iosdshid = storage(d2x(x2d(c2x(cvt))+x2d(42c)),4)
type     = storage(d2x(x2d(c2x(iosdshid))+x2d(1a)),6)
model    = storage(d2x(x2d(c2x(iosdshid))+x2d(20)),3)
man      = storage(d2x(x2d(c2x(iosdshid))+x2d(23)),3)
plant    = storage(d2x(x2d(c2x(iosdshid))+x2d(26)),2)
seqno    = storage(d2x(x2d(c2x(iosdshid))+x2d(28)),12)
say "CPC="||type||"."||model||"."||man||"."||plant||"."||seqno
```

```
# Print processor information
from pyzutil import maps

cvt = maps.ptr32(0x10)
cvtsname = maps.string(cvt + 0x154, 8, rtrim=True)
ecvt = maps.ptr32(cvt + 0x8c)
ipa = maps.ptr32(ecvt + 0x188)
ipasxnam = maps.string(ipa + 0x160, 8, rtrim=True)

print(f"SYSNAME={cvtsname} SYSPLEX={ipasxnam}")

iosdshid = maps.ptr32(cvt + 0x42C)
type = maps.string(iosdshid + 0x1A, 6, rtrim=True)
model = maps.string(iosdshid + 0x20, 3, rtrim=True)
man = maps.string(iosdshid + 0x23, 3, rtrim=True)
plant = maps.string(iosdshid + 0x26, 2, rtrim=True)
seqno = maps.string(iosdshid + 0x28, 12, rtrim=True)

print(f"CPC={type}.{model}.{man}.{plant}.{seqno}\n")
```

Control block access (cont.)

```
say " "
cvtmaxmp = c2d(storage(d2x(x2d(c2x(cvt))+x2d(1dc)),2)
cvtpccat =     storage(d2x(c2d(cvt)+x2d(2fc)),4)
say "-----------------------"
say " ID  VER  CPUID   MODEL"
say "-----------------------"
do p = 0 to cvtmaxmp
   pcca   = storage(d2x(c2d(cvtpccat)+p*4),4)
   if pcca  <> "00000000"x then do
      pccapcca = storage(d2x(c2d(pcca)),4)
      if pccapcca = "PCCA" then do
         pccavc   = storage(d2x(c2d(pcca)+4),2)
         pccacpid = storage(d2x(c2d(pcca)+6),6)
         pccamdl  = storage(d2x(c2d(pcca)+12),4)
         pccaattr = storage(d2x(c2d(pcca)+x2d(178)),1)
        sp=""
        pccaziip = bitand(pccaattr,x2c("04"))
        if pccaziip = "04"x then sp=sp"zIIP"
        pccazaap = bitand(pccaattr,x2c("01"))
        if pccazaap = "01"x then sp=sp"zAAP"
        say right(p,3," ")||"  "||pccavc " " pccacpid||,
            "  "|| pccamdl sp
      end
   end
end
```

```
cvtmaxmp = maps.uint16(cvt + 0x1DC)
cvtpccat = maps.ptr32(cvt + 0x2FC)

PCCAZIIP = 0x04   # CP is a zIIP
PCCAZAAP = 0x01   # CP is a zAAP

print(""" ID VER CPUID  MODEL
         --- --- ------ -----""")
for id in range(cvtmaxmp):
    pcaa = maps.ptr32(cvtpccat + id * 4)
    if pcaa == 0:
        continue
    if maps.string(pcaa, 4) != "PCAA":
        continue
    pccavc = maps.string(pcaa + 4, 2)
    pccacpid = maps.string(pcaa + 6, 6)
    pccamdl = maps.string(pcaa + 12, 4)
    pccaattr = maps.uint8(pcaa + 0x178)
    specialty_engine = ''
    if pccaattr & PCCAZIIP:
        specialty_engine += ' zIIP'
    if pccaattr & PCCAZAAP:
        specialty_engine += ' zAAP'
    print(f"{id:03} {pccavc}  {pccacpid} {pccamdl} {specialty_engine}")
```

QUESTIONS?
COMMENTS?
BLANK STARES?

Acknowledgements

Profuse thanks to David Crayford of Rocket Software. I had already submitted the session abstract when he mentioned on the System Z Enthusiasts Discord that he was giving a session internally. He graciously sent me his slide deck and a recording of his session, as it had no proprietary or confidential information and thus could be shared externally. I was able to integrate it (sans the live demo) with my internal storyboard and created what is the first iteration of this presentation.

At the next SHARE, I hope to be able to extend this into two parts, as there was some content I could not incorporate.

System Z Enthusiasts Discord Server

If you haven't joined, please do! The younger crowd is more comfortable with Discord than with mailing lists, and with all their infectious energy we have a great time!

There are places for serious discussions as well as off-topic threads, and there are active moderators (including myself). Come join over 1500 mainframers in lively chatter!

(Thanks to Steven Perva of Ensono for creating this server.)

Join link: https://discord.gg/system-z-enthusiasts-880322471608344597

For more information

The presentation and code samples used here are found in my Github repository

https://github.com/catherdersoftware/Python4RexxPgmr

You can email questions at mfasmcpgmr@gmail.com

You can find me on

- LinkedIn: https://www.linkedin.com/in/raymullins/
- Twitter (currently incorrectly known as X): https://twitter.com/zarchasmpgmr
- Facebook Messenger, WhatsApp, Signal, Telegram, Keybase

Your feedback is important!

SHARE
EDUCATE ‣ NETWORK ‣ INFLUENCE

Submit a session evaluation for each session you attend!

This is session 52622

www.share.org/evaluation

SHARE mobile app