Why APL is a language worth knowing

by Rodrigo Girão Serrão

FnConf 2022

About me

Rodrigo Girão Serrão

Formal education: maths

Coding in:

- Python for 9 years
- APL for 2 years

Training/teaching:

- APL (Dyalog Ltd.)
- Python, maths, etc (mathspp.com)

Why APL is a language worth knowing

A LANGUAGE THAT DOESN'T AFFECT THE WAY YOU THINK ABOUT PROGRAMMING, IS NOT WORTH KNOWING.

— Alan J. Perlis

“

A LANGUAGE THAT AFFECTS THE WAY YOU THINK ABOUT PROGRAMMING IS WORTH KNOWING.

— Rodrigo Girão Serrão, 2022?

Disclaimer

Mileage may vary!

What is APL?

What is APL?

- Programming language
  - (was “just” a mathematical notation)
- Array-oriented
- Concise
- Quirky symbols: ⍋ ⍥ ⌺ ⍠ ⍴ ⌿

What is APL?

```
      15 + 16
31
      15 - 14
1
```

What is APL?

```
      15 + 16
31
      15 - 14
1
      ⍳6
0 1 2 3 4 5
```

What is APL?

```
      15 + 16
31
      15 - 14
1
      ⍳6
0 1 2 3 4 5
```

What is APL?

```
      15 + 16
31
      15 - 14
1
      ⍳6
0 1 2 3 4 5
```

What is APL?

```
      15 + 16
31
      15 - 14
1
      ⍳6
0 1 2 3 4 5
```

What is APL?

```
10 - 5 - 2
```

??

What is APL?

```
      (10 - 5) - 2
3
```

What is APL?

```
      (10 - 5) - 2
3
      10 - 5 - 2
7
```

What is APL?

```
      (10 - 5) - 2
3
      10 - (5 - 2)
7
```

What is APL?

```
      (10 - 5) - 2
3
      10 - (5 - 2)
7
      10 - 5 - 2
7
```

Scalar functions

Scalar functions

$\iota 6$

0 1 2 3 4 5

Scalar functions

```
    ⍳6
0 1 2 3 4 5
    1+⍳6
1 2 3 4 5 6
```

Scalar functions

```
    ⍳6
0 1 2 3 4 5
    1+⍳6
1 2 3 4 5 6
    2×⍳6
0 2 4 6 8 10
```

Scalar functions

- Scalars make up all arrays
- Scalar functions act on scalars
- Good for processing all data at once

Scalar functions

```
      10 + 0 1 2 3 4 5
10 11 12 13 14 15
```

Scalar functions

```
      10 + 0 1 2 3 4 5
10 11 12 13 14 15

      0 1 2 3 4 5 + 10
10 11 12 13 14 15
```

Scalar functions

```
      10 + 0 1 2 3 4 5
10 11 12 13 14 15

      0 1 2 3 4 5 + 10
10 11 12 13 14 15

      100 0 1 × 2 3 4
200 0 4
```

Scalar functions

Power *

```
      1 2 3*2
1 4 9
```

Scalar functions

Power *

```
      1 2 3*2
1 4 9

      2*⍳6
1 2 4 8 16 32
```

Scalar functions

Residue |

```
        10|1  12  123  1234
1  2  3  4
```

Scalar functions

Residue |

```
      10|1 12 123 1234
1 2 3 4

      2|⍳5
0 1 0 1 0
```

List comprehensions

List comprehensions

```
# Square integers from 0 to 9:
```

List comprehensions

```
# Square integers from 0 to 9:
>>> squares = []
```

List comprehensions

```
# Square integers from 0 to 9:
>>> squares = []
>>> for num in range(10):
```

List comprehensions

```
# Square integers from 0 to 9:
>>> squares = []
>>> for num in range(10):
...     squares.append(num ** 2)
```

List comprehensions

```
# Square integers from 0 to 9:
>>> squares = []
>>> for num in range(10):
...     squares.append(num ** 2)
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

List comprehensions

```
# Square integers from 0 to 9:
>>> squares = []
>>> for num in range(10):
...     squares.append(num ** 2)
```

List comprehensions

```
# Square integers from 0 to 9:
```

1. Create empty result list

```
>>> for num in range(10):
...         squares.append(num ** 2)
```

List comprehensions

```
# Square integers from 0 to 9:
```

1. Create empty result list
2. Go over existing list

```
...         squares.append(num ** 2)
```

List comprehensions

```
# Square integers from 0 to 9:
```

1. Create empty result list
2. Go over existing list
3. Add modified value to result

List comprehensions

```
# Square integers from 0 to 9:
squares = []
for num in range(10):
    squares.append(num ** 2)
```

List comprehensions

```
# Square integers from 0 to 9:
squares = [num ** 2 for num in range(10)]
```

List comprehensions

- Square integers from 0 to 9:

List comprehensions

```
⍝ Square integers from 0 to 9:
 ⍳10
```

List comprehensions

```
      ⍝ Square integers from 0 to 9:
      (⍳10)*2
0 1 4 9 16 25 36 49 64 81
```

List comprehensions

```
>>> num = 42
>>> num % 10
2
```

List comprehensions

```
>>> numbers = [42, 73, 0, 16, 10]
>>> num % 10
2
```

List comprehensions

```
>>> numbers = [42, 73, 0, 16, 10]
>>> [num % 10 for num in numbers]
[2, 3, 0, 6, 0]
```

List comprehensions

```
      number ← 42
      10|number
2
```

List comprehensions

```
    numbers ← 42 73 0 16 10
    10|number
2
```

List comprehensions

```
      numbers ← 42 73 0 16 10
      10|numbers
2 3 0 6 0
```

List comprehensions

10 | number

10 | numbers

num % 10

[num % 10 for num in numbers]

List comprehensions

To write:

- Focus on transformation wanted
- Fill in the syntax

Why bother?

- Data transformation is highlighted

Boolean values

Boolean values

- Python, Haskell, ...
  - `True`, `False`
- Java, JavaScript, ...
  - `true`, `false`

Boolean values

```
      3 > 2
1 A "true"
```

Boolean values

```
      3 > 2
1 ⍝ “true”

      2 > 3
0 ⍝ “false”
```

Boolean values

Maybe weird at first..?

Actually very convenient!

`if` statements:

- If condition is true, run
- If condition is false, don't run

Fine-grained control over arrays?

- Use maths

Data-driven conditionals

Data-driven conditionals

`if` statements: “Should we do X?”

vs

DDC: “How should we do X?”

Data-driven conditionals

Car rental:

- $40/day base price
- \+ extra fees:
  - $200 if age ≥ 25
  - $500 if age ≤ 24

Data-driven conditionals

```
def rental_cost(days, age):
    price = 40 * days
```

Data-driven conditionals

```
def rental_cost(days, age):
    price = 40 * days
    if age >= 25:
```

Data-driven conditionals

```
def rental_cost(days, age):
    price = 40 * days
    if age >= 25:
        price += 200
    else:
        price += 500
```

Data-driven conditionals

```python
def rental_cost(days, age):
    price = 40 * days
    if age >= 25:
        price += 200
    else:
        price += 500
    return price
```

Data-driven conditionals

```
def rental_cost(days, age):
    base = 40 * days
    fees = 200 if age >= 25 else 500
    return base + fees
```

Data-driven conditionals

$$(40 \times days) + 200 + 300 \times age \leq 24$$

Data-driven conditionals

```
age ← 56
(40×days)+200+300×age≤24
```

Data-driven conditionals

```
age ← 56
(40×days)+200+300×0
```

Data-driven conditionals

```
age ← 56
(40×days)+200+0
```

Data-driven conditionals

```
age ← 56
(40×days)+200
```

Data-driven conditionals

```
age ← 23
(40×days)+200+300×age≤24
```

Data-driven conditionals

```
age ← 23
(40×days)+200+300×1
```

Data-driven conditionals

```
age ← 23
(40×days)+200+300
```

Data-driven conditionals

```
age ← 23
(40×days)+500
```

Data-driven conditionals

$$(40 \times days) + 200 + 300 \times age \leq 24$$

Data-driven conditionals

Car rental:

- $40/day base price
- + extra fees:
  - $200 if age ≥ 25
  - $500 if age ≤ 24

Data-driven conditionals

Car rental:

- $40/day base price
- $200 extra fees
- $300 possible surcharge (age ≤ 24)

Data-driven conditionals

```
      age ← 33
      days ← 40
      price ← (40×days)+200+300×age≤24
      price
1800
```

Data-driven conditionals

```
      age ← 33 22 45 73
      days ← 40 40 18 6
      price ← (40×days)+200+300×age≤24
      price
1800 2100 920 440
```

Data-driven conditionals

```
      age ← 33 22 45 73
      days ← 40 40 18 6
      price ← (40×days)+200+300×age≤24
      price
1800 2100 920 440
      +/price
5260
```

Data-driven conditionals

```
age = [33, 22, 45, 73]
days = [40, 40, 18, 6]
prices = []
```

Data-driven conditionals

```
age = [33, 22, 45, 73]
days = [40, 40, 18, 6]
prices = []
for a, d in zip(age, days):
```

Data-driven conditionals

```
age = [33, 22, 45, 73]
days = [40, 40, 18, 6]
prices = []
for a, d in zip(age, days):
    base = 40 * d
    fees = 200 if a >= 25 else 500
```

Data-driven conditionals

```
age = [33, 22, 45, 73]
days = [40, 40, 18, 6]
prices = []
for a, d in zip(age, days):
    base = 40 * d
    fees = 200 if a >= 25 else 500
    prices.append(base + fees)
total = sum(prices)
```

Data-driven conditionals

```
age = [33, 22, 45, 73]
days = [40, 40, 18, 6]
netted = sum(
    40 * d + 200 + 300 * (a <= 24)
    for a, d in zip(age, days)
)
```

Data-driven conditionals

```
age = [33, 22, 45, 73]
days = [40, 40, 18, 6]
netted = sum(
    40 * d + 200 + 300 * (a <= 24)
   (40×days)+200+300 × age≤24
    for a, d in zip(age, days)
)
```

Filtering list comprehensions

Filtering list comprehensions

```
# Square integers:
>>> nums = [42, 73, 0, 16, 10]
>>> [n ** 2 for n in nums]
[1764, 5329, 0, 256, 100]
```

Filtering list comprehensions

```
# Square even integers:
>>> nums = [42, 73, 0, 16, 10]
>>> [n ** 2 for n in nums if n % 2 == 0]
[1764, 0, 256, 100]
```

Filtering list comprehensions

```
     1 0 1 1 1 / 42 73 0 16 10
42 0 16 10
```

Filtering list comprehensions

```
      1 0 1 1 1 / 42 73 0 16 10
42 0 16 10

      numbers ← 42 73 0 16 10
      0=2|numbers
1 0 1 1 1
```

Filtering list comprehensions

```
      1 0 1 1 1 / 42 73 0 16 10
42 0 16 10

      numbers ← 42 73 0 16 10
      0=2|numbers
1 0 1 1 1

      (0=2|numbers)/numbers
42 0 16 10
```

Filtering list comprehensions

```
      (0=2|numbers)/numbers
42 0 16 10
```

Filtering list comprehensions

```
      (0=2|numbers)/numbers
42 0 16 10

      ((0=2|numbers)/numbers)*2
1764 0 256 100
```

Filtering list comprehensions

List comprehensions with filters:

1. Filter
2. Transform

Counting over a predicate

Counting over a predicate

```
⍝ How many 5s in here?
nums ← 5 3 7 6 4 1 9 2 5 6
```

Counting over a predicate

```
      ⍝ How many 5s in here?
      nums ← 5 3 7 6 4 1 9 2 5 6
      5=nums
1 0 0 0 0 0 0 0 1 0
```

Counting over a predicate

```
      ⍝ How many 5s in here?
      nums ← 5 3 7 6 4 1 9 2 5 6
      5=nums
1 0 0 0 0 0 0 0 1 0


      +⌿5=nums
2
```

Counting over a predicate

```
# How many 5s in here?
nums = [5, 3, 7, 6, 4, 1, 9, 2, 5, 6]
```

Counting over a predicate

```python
# How many 5s in here?
nums = [5, 3, 7, 6, 4, 1, 9, 2, 5, 6]
count = 0
for num in nums:
```

Counting over a predicate

```python
# How many 5s in here?
nums = [5, 3, 7, 6, 4, 1, 9, 2, 5, 6]
count = 0
for num in nums:
    if num == 5:
```

Counting over a predicate

```python
# How many 5s in here?
nums = [5, 3, 7, 6, 4, 1, 9, 2, 5, 6]
count = 0
for num in nums:
    if num == 5:
        count += 1
```

Counting over a predicate

```python
# How many 5s in here?
nums = [5, 3, 7, 6, 4, 1, 9, 2, 5, 6]
count = 0
for num in nums:
    count += (num == 5)
```

Counting over a predicate

```python
# How many 5s in here?
nums = [5, 3, 7, 6, 4, 1, 9, 2, 5, 6]
count = sum(num == 5 for num in nums)
```

Counting over a predicate

```
# How many 5s in here?
nums = [5, 3, 7, 6, 4, 1, 9, 2, 5, 6]
count = sum(num == 5 for num in nums)
         +/ nums = 5
```

Counting over a predicate

```
# How many values satisfy the predicate?
sum(pred(value) for value in values)
```

Recap

Recap

- Scalar functions
- Maths instead of branching
    - (data-driven conditionals)
- Compressing vs filtering in list comprehensions
- Counting idiom

References

"Why APL is a language worth knowing", https://mathspp.com/blog/why-apl-is-a-language-worth-knowing

/mathspp/talks

$$\overbrace{\underbrace{\text{rodrigo}}_{\text{name}}\text{@}\underbrace{\underbrace{\text{dyalog}}_{\text{company}}\text{.com}}_{\text{site}}}^{\text{email}}$$