Programming Lanaugages (1)
Functional Programming

Kenjiro Taura

What is functional programming? 1st attempt

- Wikipedia:

  > *...functional programming is a programming paradigm where programs are constructed by applying and composing functions.*

- ..., well, I am almost always writing and applying functions, ...

What is functional programming? 2nd attempt

- ask ChatGPT: describe the most important aspect of functional programming in less than 100 words.

  > *Functional programming's essence lies in treating computation as the evaluation of mathematical functions. It emphasizes immutability, pure functions, and higher-order functions, enabling clearer, more concise, and maintainable code. By avoiding mutable state and side effects, it enhances modularity and facilitates reasoning about program behavior.*

- is it about *"avoiding side effects"*, or more generally *avoiding certain tools that you have been taught and are now familiar with*?
- if so, why you care?

What is functional programming? My take

- it is a “way of thinking” when you solve a problem
- it formulates the solution to the problem using *recursion*
- let’s dig into it using an example

An example

- Q: write a function that computes the sum of elements in an array
- A: a "procedural" Python program

```
def sum_array(a):
    n = len(a)
    s = 0
    for i in range(n):
        s = s + a[i]
    return s
```

Thinking behind the procedural version

- well, to compute `a[0] + a[1] + ...+ a[n-1]`,
- start with `s = 0`, and
- `s = s + a[0]`
- `s = s + a[1]`
- ...
- `s = s + a[n-1]`
- now `s` should hold what we want

remember how you overcame the following confusing "equation"?

```
s = s + a[i]  # do you mean 0 = a[i] ??
```

A “functional” version

```
# a[i] + a[i+1] + ... + a[j-1]
def sum_range(a, i, j):
    if i == j:
        return 0
    else:
        return a[i] + sum_range(a, i + 1, j)

def sum_array(a):
    return sum_range(a, 0, len(a))
```

A (superficial) characteristics of the "functional" version

- *no updates* to variables (like `s = s + ...`)
- no loops

but the point is not about *avoiding* them

The thinking behind the functional version

- the observation

$$\text{sum of } \texttt{a[0:n]} = \texttt{a[0]} + \text{sum of } \texttt{a[1:n]}$$

- ... and you can compute "sum of `a[1:n]`" (almost) by a recursive call
- to be precise, you define a function to compute sum of an array range `a[i:j]` by

$$\text{sum of } \texttt{a[i:j]} = \texttt{a[i]} + \text{sum of } \texttt{a[i+1:j]}$$

- one more thing is the base case (when $\texttt{i} = \texttt{j}$, sum is zero)

Note : a few more alternatives

- $$\text{sum of } \texttt{a[i:j]} = \text{sum of } \texttt{a[i:j-1]} + \texttt{a[j-1]}$$

- $$\text{sum of } \texttt{a[i:j]} = \text{sum of } \texttt{a[i:c]} + \texttt{a[c:j]}$$

  where $\texttt{c} = \lfloor(\texttt{i} + \texttt{j})/2\rfloor$, or any value that satisfies $\texttt{i} < \texttt{c} < \texttt{j}$, for that matter

```
def sum_range(a, i, j):
    if i == j:
        return 0
    elif i + 1 == j:
        return a[i]
    else:
        c = (i + j) // 2
        return sum_range(a, i, c) + sum_range(a, c, j)
```

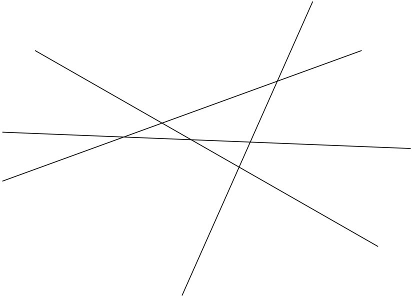
The "functional way" of problem solving

- $\approx$ solving a problem by recursive calls
- $\approx$ solving a problem by assuming solutions to "smaller" cases are known

this is very powerful because of the same reason why solving math problems using *recurrence relation (漸化式)* is very powerful

Solving problems with recurrence relation : an example

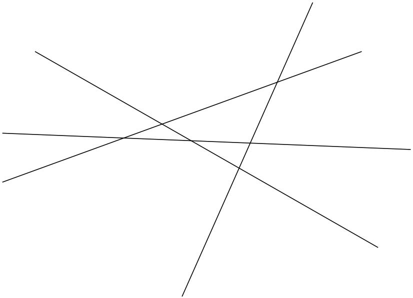
- Q: Draw n lines in a plane, in such a way that no three lines intersect at a point. How many regions do they divide the plane into?
- A: Let the number of regions a_n. Then,



Solving problems with recurrence relation : an example

- Q: Draw n lines in a plane, in such a way that no three lines intersect at a point. How many regions do they divide the plane into?
- A: Let the number of regions a_n. Then,

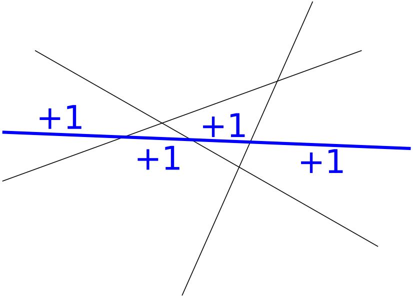
$$a_0 = 1,$$



Solving problems with recurrence relation : an example

- Q: Draw n lines in a plane, in such a way that no three lines intersect at a point. How many regions do they divide the plane into?
- A: Let the number of regions a_n. Then,

$$\begin{cases} a_0 &= 1, \\ a_n &= a_{n-1} + n \end{cases}$$



The functional thinking

1. say you are asked to find an answer to a problem (e.g., $f(n)$ or $g(a)$)
2. try to answer it, *assuming the answer to "smaller cases" are known*
3. express it using recursions

- what *"smaller"* exactly means depends on the problem
  - smaller integers (e.g., $n - 1$, $n/2$, etc.)
  - smaller arrays (e.g., $a[0 : n - 1]$, $a[1 : n]$, $a[0 : n/2]$, numbers in a less than x, etc.)
  - a child of a tree node
  - etc.

The divide-and-conquer paradigm

- a similarly powerful paradigm is the "divide-and-conquer" problem solving
- given an input X
- somehow *"divide"* X into smaller instances $X_0, X_1, \cdots$
- solve each of them using a recursion
- somehow *"merge"* them into the solution to X

One more example

- Q: define a function that, given a and n, computes a^n
  - note: Python has a builtin primitive (a `**` n) or `pow` that just does that, but here we define it without them
- A: the "procedural" version

```
def pow(a, n):
    p = 1
    for i in range(n):
        p = p * a
    return p
```

- this expresses *how* you compute a^n, step by step

A functional version

- instead ask “what is” a^n
- well,
  - base case: $n = 0 \Rightarrow 1$
  - otherwise, $a^n = a * a^{n-1}$

```
def pow(a, n):
    if n == 0:
        return 1
    else:
        return a * pow(a, n - 1)
```

A smarter version

```
def pow(a, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        p = pow(a, n // 2)
        return p * p
    else:
        return a * pow(a, n - 1)
```