Módulo 02 - Geradores

Tópicos em Linguagens

https://github.com/fsantanna-uerj/LPX/

Francisco Sant'Anna

francisco@ime.uerj.br

1. Geradores

Tópicos em Linguagens

`https://github.com/fsantanna-uerj/LPX/`

Francisco Sant'Anna

`francisco@ime.uerj.br`

Geradores

Gerador (ciência da computação)

15 línguas

Artigo Discussão Ler Editar Ver histórico

Origem: Wikipédia, a enciclopédia livre.

> ***Nota:*** *Para outros significados de Generator, veja Generator.*

Em ciência da computação, um **gerador** (em inglês, *generator*) é um procedimento especial que pode ser usado para controlar iteradores de *loops*. Um gerador é muito similar para funções que retornam *arrays* (ou vetores), geradores podem ter parâmetros, que também podem ser chamados e geram uma sequência de valores. Entretanto, em vez de construir uma sequência que contenha todos os valores e os retornam de uma só vez, um gerador utiliza a palavra-chave *yield* para retornar os valores um de cada vez, que utiliza menos memória e permite o processamento de poucos valores rapidamente. Um gerador é uma função mas comporta-se como um iterador.

As primeiras aparições de geradores foram na CLU em 1975, [1] e atualmente são encontrados facilmente em *softwares* em linguagem de programação Python e C#. (em CLU e C# generators são chamados de iteradores).

Geradores

- Uma função que representa uma coleção, podendo retornar múltiplas vezes **sem terminar** (mantendo o seu estado interno).
- Quando chamada, retorna o próximo elemento da coleção, ou sinaliza o seu término.

2. Geradores em Python

Tópicos em Linguagens

`https://github.com/fsantanna-uerj/LPX/`

Francisco Sant'Anna

`francisco@ime.uerj.br`

Geradores em Python

```
for v in iter([1,2,3]):
    print(v)
```

```
def f ():
    yield 1
    yield 2
    yield 3

for v in iter(f()):
    print(v)
```

```
it = iter(f())
print(it)
print(next(it))
print(next(it))
print(next(it))
print(next(it))
```

02-gen/01-python.py

- Iteração
  - `for`, `next->next->next`
- Gerador é um Iterador
  - objeto com next
  - next acorda gerador
  - yield “produz” um valor
- f vs f() vs next(f())

Retorna todos os naturais

```
def Nats ():
    n = 0
    while True:
        yield(n)
        n += 1

for v in Nats():
    print(v)
```

02-gen/02-nats.py

Exercício 2.2.1

- Crie um **iterador** em Python que retorna todos os números pares.
- Crie um **gerador** em Python que retorna todos os números pares.
- Faça uma discussão entre as diferenças fundamentais entre as duas abordagens.

```
for n in iter(Pares()):
    print(n)

# saida
2
4
6
...
```

3. Geradores em Lua

Tópicos em Linguagens
`https://github.com/fsantanna-uerj/LPX/`

Francisco Sant'Anna
`francisco@ime.uerj.br`

Geradores em Lua

```lua
for i,v in ipairs({10,20,30}) do
    print(v)
end
```

```lua
function f ()
    coroutine.yield(1,10)
    coroutine.yield(2,20)
    coroutine.yield(3,30)
end

for i,v in coroutine.wrap(f) do
    print(i,v)
end
```

```lua
it = coroutine.wrap(f)
print(it)
print(it())
print(it())
print(it())
print(it())
print(it())
```

02-gen/03-lua.lua

- Iteração
  - `for`, `it()->it()`
- Gerador é um Iterador
  - **wrap** empacota estado em uma closure (stateful)
  - chamada acorda gerador
  - yield “produz” um valor
- f vs wrap(f) vs wrap(f)()

Exercício 2.3.1

- Crie um **gerador** em Lua que abre o arquivo "teste.txt" e retorna todos os seus caracteres, um a um.
- Use as seguintes funções:
  - io.open, f:read(1), f:close

```
for c in coroutine.wrap(F):
    print(c)

# saida
a
b
c
...
```

Co-rotinas em Lua

```
function f ()
    coroutine.yield(1,10)
    coroutine.yield(2,20)
    coroutine.yield(3,30)
    --return nil
end

co = coroutine.create(f)
print(coroutine.resume(co))
print(coroutine.resume(co))
print(coroutine.resume(co))
print(coroutine.resume(co))
--print(coroutine.resume(co))
```

- **create**: cria coro a partir de função
- **resume**: acorda coro a partir do último ponto de parada (início ou **yield**)
- **yield**: dorme coro e retorna para último **resume**

02-gen/03-lua.lua

Lua - wrap

```
function wrap (f)
    local co = coroutine.create(f)
    return function ()
        local _,v = coroutine.resume(co)
        return v
    end
end

function f ()
    coroutine.yield(10)
    coroutine.yield(20)
    coroutine.yield(30)
end

for v in wrap(f) do
    print(v)
end
```

- **wrap**: empacota **create** e **resume** em uma closure
- ignora status e erros

02-gen/04-wrap.lua

4. Exercício

Tópicos em Linguagens
`https://github.com/fsantanna-uerj/LPX/`

Francisco Sant'Anna
`francisco@ime.uerj.br`

Exercício

- “The Paradigms of Programming”
- Turing Award 1978 - Robert W. Floyd

- *Read lines of text, until a completely blank line is found. Eliminate redundant blanks between the words. Print the text, thirty characters to a line, without breaking words between lines.*

Again, take this simple-looking problem:

> Read lines of text, until a completely blank line is found. Eliminate redundant blanks between the words. Print the text, thirty characters to a line, without breaking words between lines.

Because both input and output are naturally expressed using multiple levels of iteration, and because the input iterations do not nest with the output iterations, the problem is surprisingly hard to program in most programming languages [14]. Novices take three or four times as long with it as instructors expect, ending up either with an undisciplined mess or with a homemade control structure using explicit incrementations and conditional execution to simulate some of the desired iterations.

The problem is naturally formulated by decomposition into three communicating coroutines [4], for input, transformation, and output of a character stream. Yet, except for simulation languages, few of our programming languages have a coroutine control structure adequate to allow programming the problem in a natural way.

When a language makes a paradigm convenient, I will say the language *supports* the paradigm. When a language makes a paradigm feasible, but not convenient, I will say the language *weakly supports* the paradigm. As the two previous examples illustrate, most of our languages only weakly support simultaneous assignment, and do not support coroutines at all, although the mechanisms required are much simpler and more useful than, say, those for recursive call-by-name procedures, implemented in the Algol family of languages seventeen years ago.

Exercício

- *Read lines of text, until a completely blank line is found. Eliminate redundant blanks between the words. Print the text, thirty characters to a line, without breaking words between lines.*

- Entrada em um bloco de linhas (parágrafo) por vez
- Saída em um linha de 30 colunas por vez
- As linhas não devem ter espaços redundantes

Exercício

- Criar um iterador (s/ gerador) para tal problema.
- Iremos ver um esboço do iterador.
- Iremos ver uma solução alternativa usando gerador.

- lazy
- arvores
- infinito