quinta-feira, 1 de abril de 2021 07:31

Paradigmas de programação

Imperativo

Repetições explícitas com os loops for, while, do while

Transparente* → sabe exatamente o que está acontecendo

Tempo verbal imperativo → ordem definida deve ser seguida
↳ mandar, sequência
↳ faça isso, depois isso, depois aquilo
+ natural para o ser humano

+ popular
presente em praticamente todas as linguagens de programação

+ fácil de implementar procedimentos complexos

mais baixo nível
↳ proximidade da máquina

pode afetar e ser afetado por variáveis externas e iterações anteriores

permite efeitos colaterais

+ flexível, permitindo inclusive resolver problemas de simulação

Funcional

Repetição implícita com a aplicação de funções de forma repetida

Claro* → mais fácil de entender

O poder de abstração das funções
↳ isolar o que é mais importante em detrimento daquilo que supérfluo (ou complexo)

é cada vez mais popular

suporte em muitas linguagens
↳ necessita ter funções que suportem funções como argumento

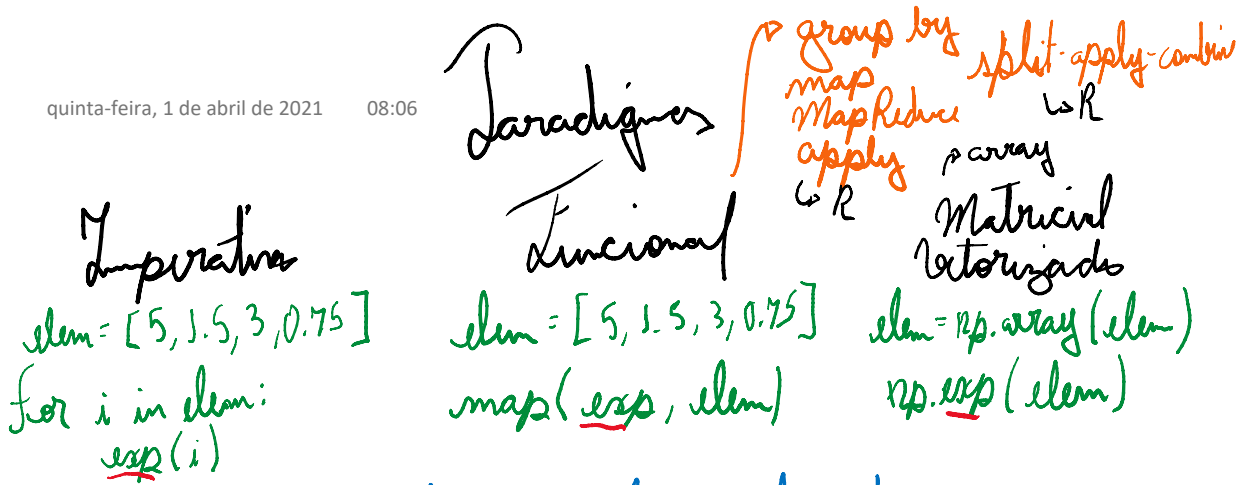
+ fácil implementar procedimentos de baixa e média complexidade

mais alto nível
+ próximo ser humano

não permite efeitos colaterais

Todas as iterações devem ser independentes
↳ facilita paralelizar procedimentos
executar em múltiplos núcleos e computadores (distribuídos)

quinta-feira, 1 de abril de 2021 08:06



Exponencial em elementos

```python
#************************************#
#--------- Imperativo ----------------#
from math import exp
elem = [5, 1.5, 3, 0.75]
# versão 1
elem_exp = []
for i in elem:
    elem_exp.append(exp(i))

# versão 2: compreensão listas
elem_exp = [exp(i) for i in elem]
```

```python
#************************************#
#--------- Funcional ----------------#
from math import exp
elem = [5, 1.5, 3, 0.75]
elem_exp = list(map(exp, elem))
```

```python
#************************************#
#--------- Vetorizado ----------------#
import numpy as np
elem = np.array([5, 1.5, 3, 0.75])
elem_exp = np.exp(elem)
```

Condicional em elementos

```python
#************************************#
#--------- Imperativo ----------------#
niveis = [5, 1.5, 3, 0.75]
acao = []
for nivel in niveis:
    if nivel >= 2:
        acao.append(nivel)
```

```python
#************************************#
#--------- Funcional ----------------#
# map(função, elementos)
niveis = [5, 1.5, 3, 0.75]
# Versão 1
def decisao(nivel):
    if nivel >= 2: return nivel

acao = list(filter(decisao, niveis))

# Versão 2
# lambda para criar funções anonimas
list(filter(lambda x: x >= 2, niveis))
```

```python
#************************************#
#--------- Vetorizado ----------------#
import numpy as np
niveis = np.array([5, 1.5, 3, 0.75])
acao = niveis[niveis >= 2]
```

Ajuste de modelo para grupos de elementos

```python
# 3. Operação mais complexa:
# Ajuste de modelos para grupos de elementos
import pandas as pd
import statsmodels.formula.api as smf
proj = [1, 1, 1, 1, 2, 2, 2, 2, 2,
        3, 3, 3, 3]
x = [1, 2, 3, 4, 1, 2, 3, 4, 5,
     1, 2, 3, 4]
y = [1, 2, 2, 3, 8, 6, 5, 3, 2,
     2, 5, 8, 9]
lista = zip(proj, x, y)
col = ['proj', 'x', 'y']
d = pd.DataFrame(lista, columns = col)
```

```python
#************************************#
#--------- Imperativo ----------------#
uni_proj = d['proj'].unique()
coef = []
for i in uni_proj:
    d_i = d.loc[d['proj'] == i]
    ml = smf.ols('y ~ x', d_i).fit()
    coef.append([i, ml.params[0], ml.params[1]])

col = ['proj', 'b0', 'b1']
df_coef = pd.DataFrame(coef, columns = col)
```

```python
#************************************#
#--------- Funcional ----------------#
def ajuste(df):
    ml = smf.ols('y ~ x', df).fit()
    return ml.params

df_coef = d.groupby('proj').apply(ajuste)
```

```python
#************************************#
#--------- Vetorizado ----------------#
# ?
```