Motivação

Existem várias formas de entender **se um código** é considerado **bom ou ruim**.

Algumas **características** são consenso dentre os programadores, como: **Menos linhas de código** e **utilização de boas práticas** estabelecidas pela comunidade de determinada linguagem.

Essas características ajudam a gente a gerar **código eficiente**.

PEP - Python Enhancement Proposal, A Guideline

List Comprehension

cubos
// academy //

List Comprehension

- Sintaxe usada para **criar listas baseadas em listas existentes**
- Abordagem elegante
- **Mais eficiente**: Python aloca memória primeiro, em vez de alocar na execução.
- Poder reescrever um **loop for** em uma **única linha de código**

Sintaxe básica

```
nova_lista = [expressao for item in lista]
```

List Comprehension

Sem List Comprehension

```python
numero_ao_quadrado = []

for numero in range(10):
  numero_ao_quadrado.append(numero*numero)

# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Com List Comprehension

```python
numero_ao_quadrado =
[numero * numero for numero in range(10)]

# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Usando Condicionais

Podem utilizar **expressões condicionais** para criar ou modificar listas. **Apenas uma condição**:

```
preco_inicial = [1.25, -9.45, 10.22, 3.78,
-5.92, 1.16]

precos_validados = [item if item > 0 else 0
for item in preco_inicial]

# [1.25, 0, 10.22, 3.78, 0, 1.16]
```

Usando Condicionais

Podem utilizar **expressões condicionais** para criar ou modificar listas. **Várias condições**:

```
precos_validados = [1.25, 0, 10.22, 3.78, 0, 1.16]

entre_zero_dez = [item for item in precos_validados
if item >= 0 if item < 10]

# [1.25, 0, 3.78, 0, 1.16]
```

Loops Aninhados

Para percorrer duas listas podemos utilizar **loops aninhados**:

```python
turma1 = ["Amanda", "Arthur", "Ana", "Aristides"]
turma2 = ["Beatriz", "Bernardo", "Bruna", "Benício"]

combinacoes = [(aluno1, aluno2) for aluno1 in turma1
for aluno2 in turma2]

print(combinacoes)

#[('Amanda', 'Beatriz'),
# ('Amanda', 'Bernardo'),
# ('Amanda', 'Bruna'),
# ('Amanda', 'Benício'),
# ('Arthur', 'Beatriz'),
# ('Arthur', 'Bernardo'),
# ('Arthur', 'Bruna'),
# ('Arthur', 'Benício'),
# ('Ana', 'Beatriz'),
# ('Ana', 'Bernardo'),
# ('Ana', 'Bruna'),
# ('Ana', 'Benício'),
# ('Aristides', 'Beatriz'),
# ('Aristides', 'Bernardo'),
# ('Aristides', 'Bruna'),
# ('Aristides', 'Benício')]
```

Exercícios

cubos academy

Funções Lambda

cubos academy

Motivação

Muitas vezes na construção de um código precisamos **criar funções** e **nem sempre** essa função precisa **ser reaproveitada** em outros lugares do código.

Essa função **se aplica apenas para aquele contexto**. Como por exemplo quando utilizamos métodos, tipo o **apply** em Dataframes.

Para esse caso onde temos uma **função simples sem reaproveitamento** existe uma **sintaxe mais simples**, chamada **Lambda Functions**.

Funções lambda

- Funções **Anônimas**
- Não precisam ser definidas e nem nomeadas
- Podem ser utilizadas em qualquer lugar onde você usaria uma função padrão

Uma função lambda pode receber **qualquer número de argumentos**, mas pode ter **apenas uma instrução ou retorno**.

```
lambda arg1, arg2, arg3: arg1 + arg2 + arg
```

Comparanções de Sintaxe

Funções lambda são mais **concisas** quando comparadas a funções clássicas definidas pelo usuário.

```
def promocao(preco):
  return preco-(preco*0.2)

carros['precos'] =
carros['precos'].apply(promocao)
```

```
carros['precos'] =
carros['precos'].apply(lambda x: x-(x*0.2))
```

Exercícios

cubos academy

Complementos à função lambda

cubos academy

map()

O map **retorna um objeto** depois de **aplicar uma regra** por meio de uma **função** em **cada item** de uma **lista existente**. Utilizado para gerar novas listas, ou manipular valores.

Importante: É necessário transformar o resultado em uma lista usando list()

```python
cesta_frutas = [['Mexerica', '200g'], ['Maçã', '150g'], ['Manga',
'125g'], ['Limão', '300g']]

frutas_na_cesta = list(map(lambda x: f"{x[1]} de {x[0]}", cesta_frutas))

print(frutas_na_cesta)
# ['200g de Mexerica', '150g de Maçã', '125g de Manga', '300g de Limão']
```

map()

É possível aplicar o map em **mais de uma lista ao mesmo tempo**

Importante: É necessário transformar o resultado em uma lista usando list()

```python
cesta_frutas = ['Mexerica', 'Maçã', 'Manga', 'Limão', 'Morango', 'Acerola']
info_frutas = [['Mexerica', '200g'], ['Maçã', '150g'], ['Manga', '125g'], ['Limão', '300g']]

alimentos_nas_cestas = map(lambda fruta, info: info[0] == fruta and f"{info[1]} de {info[0]}", cesta_frutas, info_frutas)

alimentos_nas_cestas = list(alimentos_nas_cestas)

print(alimentos_nas_cestas)
# ['200g de Mexerica', '150g de Maçã', '125g de Manga', '300g de Limão']
```

filter()

Filtra itens em uma **lista**, **sem alterar a lista** utilizada.

```python
info_frutas = [['Mexerica', '200g'], ['Maçã', '0g'],
['Manga', '0g'], ['Limão', '300g']]

frutas_com_peso = list(filter(lambda info: info[1] != '0g'
, info_frutas))

print(frutas_com_peso)
# [['Mexerica', '200g'], ['Limão', '300g']]
```

apply()

O método do pandas também pode receber uma **lambda function como argumento**.

```
# Transformando a lista em um DataFrame
info_frutas = pd.DataFrame([['Mexerica', '200g'], ['Maçã', '100g'],
['Manga', '150g'], ['Limão', '300g']])

# Usando apply para transformar as gramas em um inteiro
info_frutas[1] = info_frutas[1].apply(lambda x int(x.split('g')[0]))
```

reduce()

O reduce é um método **muito utilizado** por ele ser **incremental**, isso significa que ele **guarda um valor** que pode ser usado em cada passo do loop, **retornando um único valor no final**. Útil para operações utilizando todos os itens de uma lista.

Importante: Para usar esse método temos que importá-lo da biblioteca functools

```python
from functools import reduce

numeros = [200, 300, 400, 500, 600]

peso_total = reduce(lambda acumulado, atual: acumulado + atual, numeros, 0)

print(peso_total)
#2000
```

reduce()

O reduce também pode ser aplicado em **dicionários**. Depois da lista dentro do reduce podemos passar um **terceiro argumento** que será o **valor inicial do acumulador**

Importante: Para usar esse método temos que importá-lo da biblioteca functools

```
from functools import reduce

alunas = {
    'Maria': 9.9,
    'Eduarda': 7.8,
    'Lorena': 8.9,
    'Rafaela': 8.2
}

soma_notas = reduce(lambda acumulado, nome: acumulado + alunas[nome],
alunas, 0)

media_turma = soma_notas/len(alunas)

print(media_turma)
#8.7
```

reduce()

Outro **exemplo** usando **dicionários** e o **terceiro argumento**.

Importante: Para usar esse método temos que importá-lo da biblioteca functools

```python
from functools import reduce

notas = [
    {
        'materia': 'matematica',
        'nota': 9.7,
    },
    {
        'materia': 'quimica',
        'nota': 6.0,
    },
    {
        'materia': 'portugues',
        'nota': 7.5,
    }
]

soma_notas = reduce(lambda acumulado, item: acumulado + item['nota'], notas, 10)

print(soma_notas)
#23.2 + 10 = 33.2
```

Exercícios

cubos academy

Helena Guimarães
Professora Cubos Academy

cubos
// academy //

www.cubos.academy