Conceitos de Programação Funcional

Neste capítulo vamos abordar um conjunto de técnicas diferente do que temos utilizado até o momento para planejar nossos programas. Para que isso faça sentido, vamos começar entendendo um pouco melhor de que maneiras podemos planejar um programa.

1. Paradigmas de programação

Um conceito bastante importante em programação é a ideia de **paradigmas de programação**. Um paradigma é uma forma diferente de pensar o seu programa.

Todos os nossos programas até agora consistem de uma sequência de **instruções** para o computador. Essas instruções são executadas sequencialmente. A ideia de um programa como um conjunto sequencial de instruções é conhecido como **paradigma imperativo** ou **programação imperativa**, pois o programa pode ser visualizado como um conjunto de verbos no imperativo.

Na prática, nossos programas não são 100% sequenciais, e conseguimos ramificar o fluxo de execução através de condicionais ou malhas de repetição. O uso dessas técnicas foi uma evolução na forma de programar e é conhecido como **programação estruturada**.

Indo um pouco além, dificilmente escrevemos blocos muito grandes de código de uma vez. Ao longo deste curso, criamos o hábito de modularizar os nossos programas encaixando pequenos pedaços de lógica dentro de funções, que agem como *mini-programas*, e o nosso programa se desenvolve através da interação entre essas funções. Essa forma de programação é conhecida como **programação procedural**. A programação procedural é uma forma mais específica de programação imperativa.

A **programação orientada a objetos** é outro paradigma de programação, onde o foco do programa está na modelagem dos nossos programas como resultado da interação entre diferentes objetos, que possuem seu comportamento ditado por suas respectivas classes. Porém, dentro das classes nós desenvolvemos *métodos*, que são bastante semelhantes às funções que já conhecemos, e portanto, essa também é uma forma de programação imperativa.

Um ponto importante sobre a relação entre paradigmas e linguagens de programação é que para programar utilizando um certo paradigma, muitas vezes será necessário que a linguagem forneça ferramentas específicas para tal. Por exemplo, é difícil aplicar o paradigma orientado a objeto utilizando uma linguagem que não suporte conceitos importantes deste paradigma, como os próprios objetos, com seus atributos e métodos.

O Python oferece ferramentas para auxiliar na programação procedural, na programação orientada a objeto e na programação funcional, que já discutiremos.

Existe todo um conjunto de formas diferentes de programar conhecido como **programação declarativa**, onde o programador irá se preocupar mais em descrever os resultados desejados do que em enfileirar instruções. O foco está em **o que** deve ser feito, e não em **como** deve ser feito. Uma forma de programação declarativa é a **programação funcional**.

2. O que é programação funcional

Assim como na programação procedural, nossos programas serão modularizados em funções, pequenos pedaços reutilizáveis de código que podem receber dados de entrada (parâmetros) e fornecer resultados de sua computação (retorno).

Uma das motivações para a adoção da programação funcional é aumentar o **determinismo** de nossos programas, isto é, tornar a execução dos programas mais previsíveis. Programas escritos de maneira imperativa tendem a ter efeitos colaterais causados por seu **estado**. O estado de um programa é definido pelo valor de todas as suas variáveis em um dado momento.

Quando um programa se torna muito grande, com diversas funções alterando o conteúdo de diferentes variáveis e objetos, torna-se cada vez mais perigoso que o programador perca o controle sobre o estado do programa, sendo possível que certos conjuntos de valores alterem o comportamento de outras funções e resultem em erros de computação.

2.1. Funções puras

Para evitar o não-determinismo, um conceito importante é o de **funções puras**. Uma função pura não possui **efeitos colaterais**, ou seja, seu funcionamento não irá alterar conteúdo na memória (ex: variáveis) ou dispositivos de entrada e saída. Isso traz algumas vantagens:

Se nenhum parâmetro da função causa efeitos colaterais, então a função sempre apresentará o mesmo resultado ao receber os mesmos argumentos.
Se não houver dependência entre os dados de duas funções puras, elas podem ser executadas em paralelo sem causar problemas de concorrência.
Se o retorno de uma função sem efeitos colaterais não está sendo usado, ela pode ser removida sem afetar o restante do programa.

Vejamos alguns exemplos:

```
def funcao_pura(numeros: list):
    soma = 0
    for n in numeros:
        soma += n
    return soma

def funcao_impura(numeros: list):
    soma = 0
    for idx in range(len(numeros)):

        if type(numeros[idx]) != int:
            numeros[idx] = int(numeros[idx])

        soma += numeros[idx]
        idx += 1
    return soma

entrada = [1, 2.5, 3, 4.5]

funcao_pura(entrada)
print(entrada)
funcao_impura(entrada)
print(entrada)
```

Note que no exemplo acima não utilizamos o retorno de nenhuma das funções. A `funcao_pura` poderia facilmente ser removida e nada mudaria, pois ela não provocou efeitos colaterais. Já a `funcao_impura` afeta o conteúdo da lista que foi passada, o que significa que sua ausência no programa alteraria o resultado.

2.2. Funções de primeira classe e funções de alta ordem

Dizemos que algumas linguagens tratam funções como *cidadãs de primeira classe*. Isso significa que nessa linguagem funções podem:

1. ser atribuídas para variáveis ou guardadas em estruturas de dados
2. ser passadas como parâmetros para outras funções
3. ser retornadas por outras funções

É como se funções fossem variáveis como qualquer outra, de um tipo específico "função".

Essas características permitem a criação de **funções de alta ordem**, que são funções que recebem pelo menos uma função como parâmetro e/ou retornam uma função. Vejamos alguns exemplos:

1. Atribuindo função para uma variável em Python

```
def funcao():
    print('olá mundo')

x = funcao
print('Tipo da variável x:', type(x))
x()
```

Saída na tela:

```
Tipo da variável x: <class 'function'>
olá mundo
```

2. Passando uma função como parâmetro para outra função

```
def soma(a, b):
    return a + b

def multiplicacao(a, b):
    return a * b

def cumulativo(inicial, quantidade, operacao):
    contador = 1
    acumulado = inicial
    while contador <= quantidade:
        acumulado = operacao(acumulado, contador)
        contador += 1
    return acumulado

somatorio = cumulativo(0, 5, soma)
fatorial = cumulativo(1, 5, multiplicacao)
print(f'Somatório de 1 a 5: {somatorio} | Fatorial de 5: {fatorial}')
```

Saída na tela:

```
Somatório de 1 a 5: 15 | Fatorial de 5: 120
```

3. Retornando uma função

```
def soma(a, b):
    return a + b

def subtracao(a, b):
    return a - b

def multiplicacao(a, b):
    return a * b

def divisao(a, b):
    return a / b

def operador_para_funcao(operador):
    if operador == '+':
        return soma
    elif operador == '-':
        return subtracao
    elif operador == '*':
        return multiplicacao
    else:
        return divisao

x = operador_para_funcao('*')
print(x(5, 2))
```

2.3. Clausura

Uma possibilidade criada por todas essas estratégias é a de uma **closure** (*clausura* em algumas traduções em português), onde uma função pode ser usada para criar outra função junto de um **ambiente**. Soa complicado, mas através de um exemplo podemos entender facilmente o que está acontecendo:

```
def cria_somas(x):
    def soma(y):
        return x + y
    return soma

incremento = cria_somas(1)
decremento = cria_somas(-1)

print(incremento(5))
print(incremento(10))
print(incremento(15))

print(decremento(5))
print(decremento(10))
print(decremento(15))
```

Saída na tela:

```
6
11
16
4
9
14
```

A função "mais interna" possui um valor que foi recebido como parâmetro pela função mais externa. Quando chamamos a função `cria_somas` passando 1, a função interna foi definida como sempre retornando seu parâmetro + 1. Sendo assim, ao pegarmos essa função retornada, ela permanentemente retornará parâmetro + 1. No segundo caso passamos -1. Neste caso, a função mais interna foi criada com x valendo -1. Logo, ao pegarmos essa função, ela sempre irá retornar seu parâmetro - 1.

2.4. Funções anônimas

Outro conceito bastante comum na programação funcional é o de **funções anônimas**, também conhecidas como **funções lambda**. Elas são funções que não necessariamente precisam ser declaradas - no caso do Python, utilizando a palavra `def` - e atribuídas para um nome. A sintaxe para criar uma função lambda em Python é:

```
lambda parametro1, parametro2, ... : expressao
```

Assim como em outras funções, a lambda pode ser chamada passando parâmetros entre parênteses e sua expressão será retornada. Não utilizamos a palavra `return`, já é subentendido que o resultado da lambda será retornado.

É comum utilizarmos funções lambda para rapidamente passar um parâmetro para uma função ou obter um retorno de uma função. Vejamos alguns exemplos anteriores refeitos utilizando lambdas:

```
#Exemplo 1
def cumulativo(inicial, quantidade, operacao):
    contador = 1
    acumulado = inicial
    while contador <= quantidade:
        acumulado = operacao(acumulado, contador)
        contador += 1
    return acumulado
```

```
somatorio = cumulativo(0, 5, lambda x, y: x + y)
fatorial = cumulativo(1, 5, lambda x, y: x * y)
print(f'Somatório de 1 a 5: {somatorio} | Fatorial de 5: {fatorial}')

#Exemplo 2
def cria_somas(x):
    return lambda y: x + y

incremento = cria_somas(1)
decremento = cria_somas(-1)

print(incremento(5))
print(incremento(10))
print(incremento(15))

print(decremento(5))
print(decremento(10))
print(decremento(15))
```

2.5. Imutabilidade

Outro ponto bastante explorado pela programação funcional é tentar evitar o uso de estruturas mutáveis. Isso ocorre para evitar efeitos colaterais, como problemas de **concorrência** na execução paralela de diferentes trechos de código que poderiam afetar uma mesma estrutura.

Em vez de atualizarmos diferentes variáveis para ir salvando o estado de operações, iremos compor chamadas de funções de alta ordem e funções recursivas, e as informações irão fluir através de seus parâmetros e retornos.

Veremos logo mais algumas funções de alta ordem *tradicionais* e como elas nos ajudam a evitar a mutabilidade.

2.6. Recursão

Laços de repetição tradicionais, como o `for`, apresentam pelo menos dois possíveis problemas.

O primeiro deles é a necessidade de mutabilidade e variáveis de controle. Frequentemente, controlamos nossas repetições incrementando algum tipo de variável e testando seu valor.

O segundo é a própria legibilidade do loop: parte do código gerado é para controlar as repetições, e pode se distanciar um pouco da definição do problema em si.

Você já estudou esse conceito antes, mas vamos relembrar: recursão é quando uma função é capaz de chamar a si mesma. Tipicamente funções recursivas possuem 1 ou mais **casos base**, ou seja, casos onde nenhuma computação será feita e o resultado será retornado de maneira imediata, e o **caso geral** ou **caso recursivo**, onde o problema será subdividido em sucessivas chamadas para a função, variando seus parâmetros até que um caso base seja atingido.

Um exemplo clássico das aulas de programação é a sequência de Fibonacci. Uma das formas de definir essa sequência é a seguinte:

> F(0) = 1
> F(1) = 1
> F(n) = F(n-1) + F(n-2), se n > 1

Compare agora um código em loop (chamado de código **iterativo**) com um código **recursivo**. Note que no recursivo não utilizamos qualquer tipo de dado mutável. Observe também qual dos códigos lembra mais a definição original da sequência.

```
def fib_iterativo(n):
    n1 = 0
    n2 = 1
    contador = 0
    while contador < n:
        n1, n2 = n2, n1+n2
        contador+=1
```

```
        return n2

def fib_recursivo(n):
    if n == 0 or n == 1:
        return 1
    else:
        return fib_recursivo(n-1) + fib_recursivo(n-2)
```

Uma desvantagem da recursão é a possibilidade de muitos cálculos repetidos serem realizados. Para calcular F(5), teremos F(4) + F(3). Mas no F(4), o F(3) irá aparecer novamente.

Algumas linguagens oferecem um recurso chamado de **tail recursion**, ou **recursão de cauda**. Nelas, o resultado de chamadas recursivas recentes é temporariamente armazenado e pode ser reutilizado em outras chamadas. O Python **NÃO** possui recursão de cauda nativamente, mas é possível manipularmos os parâmetros de nossas funções para obter esse tipo de recursão na prática. Observe o código abaixo:

```
def fib_cauda(n, n1 = 1, n2 = 1):
    if n == 0:
        return n1
    if n == 1:
        return n2
    return fib_cauda(n - 1, n2, n1 + n2)
```

Em cada passo recursivo, atualizamos n2 com a próxima soma, o valor anterior de n2 passa para n1, e o nosso n decrementa rumo ao caso base. Ao contrário da chamada recursiva anterior, observe que não estamos mais "ramificando" nossas chamadas recursivas, e as chamadas estão sendo resolvidas de maneira mais linear, com cada passo aproveitando os dois resultados anteriores.

2.7. Funções de alta ordem em coleções

Qualquer função que receba ou retorne uma função é considerada uma função de alta ordem. Mas três funções de alta ordem são consideradas bastante importantes porque permitem realizar diversas operações úteis sobre coleções (listas, arrays, tuplas, etc). Elas resolvem problemas tradicionais envolvendo as coleções, mas sem a necessidade de utilizar loops. Além disso, elas sempre irão retornar uma nova lista ou um valor, não alterando o conteúdo da função original. Elas são as funções `map`, `filter` e `reduce`.

2.7.1. Map

A função `map` recebe uma função e uma coleção. Ela irá aplicar a função recebida sobre cada um dos elementos da coleção, retornando uma nova coleção com os retornos de cada uma dessas chamadas.

Em Python, a função `map` retorna um **iterador**, então cabe a nós convertê-lo para uma estrutura caso necessário, como lista ou tupla.

```
# exemplo 1 - convertendo todo o conteúdo de uma tupla para float
tupla_str = ('1.0', '3.7', '5.4')
tupla_float = tuple(map(float, tupla_str)) # função: float; coleção: tupla_str
print(tupla_str, tupla_float)

# exemplo 2 - elevando a 2 todos os elementos de uma lista usando uma função já existente
def quadrado(x):
    return x ** 2
numeros = [1, 2, 3, 4]
numeros_quadrados = list(map(quadrado, numeros)) # função: quadrado; coleção: numeros
print(numeros, numeros_quadrados)

# exemplo 3 - elevando a 3 todos os elementos de uma lista usando um lambda
numeros = [1, 2, 3, 4]
```

```
numeros_cubo = list(map(lambda x: x**3, numeros))
print(numeros, numeros_cubo)
```

Saída na tela:

```
('1.0', '3.7', '5.4') (1.0, 3.7, 5.4)
[1, 2, 3, 4] [1, 4, 9, 16]
[1, 2, 3, 4] [1, 8, 27, 64]
```

A função `map` é bastante conhecida no meio da programação funcional e é oferecida em diversas linguagens. Por exemplo, em JavaScript - uma linguagem multiparadigma como o Python - é comum o uso de `map`, bem como o das outras duas funções que estudaremos.

Apesar do Python trazer a função implementada, é possível reproduzir a funcionalidade do `map` utilizando uma compreensão de lista ou uma expressão geradora. O resultado é equivalente: evitamos mutabilidade e efeitos colaterais, geramos uma coleção nova a partir da antiga, mas o código fica mais *idiomático*.

```
# exemplo 1 - convertendo todo o conteúdo de uma tupla para float
tupla_str = ('1.0', '3.7', '5.4')
tupla_float = tuple(float(x) for x in tupla_str)
print(tupla_str, tupla_float)

# exemplo 2 - elevando a 2 todos os elementos de uma lista usando uma função já existente
def quadrado(x):
    return x ** 2
numeros = [1, 2, 3, 4]
numeros_quadrados = [quadrado(x) for x in numeros]
print(numeros, numeros_quadrados)

# exemplo 3 - elevando a 3 todos os elementos de uma lista sem usar função pronta
numeros = [1, 2, 3, 4]
numeros_cubo = [x**3 for x in numeros]
print(numeros, numeros_cubo)
```

2.7.2. Filter

A função `filter` também recebe uma função que deve retornar um booleano e uma coleção. Ela irá conter apenas os elementos da coleção que provocaram valor `True` na função passada.

```
# exemplo 1 - detectando pares em uma lista usando função pronta
def eh_par(x):
    return x % 2 == 0
numeros = [3, 6, 4, 8, 7, 9, 2, 5]
pares = list(filter(eh_par, numeros)) # função: eh_par; coleção: numeros
print(pares)

# exemplo 2 - detectando negativos em uma lista usando lambda
numeros = [5, -3, 1, 4, 7, -8, -2]
negativos = list(filter(lambda x: x < 0, numeros))
print(negativos)
```

Saída na tela:

```
[6, 4, 8, 2]
[-3, -8, -2]
```

Assim como no caso do `map`, podemos reproduzir a funcionalidade do `filter` de maneira mais *pythonica* utilizando compreensão de lista ou expressão geradora:

```
# exemplo 1 - pares usando função pronta
def eh_par(x):
    return x % 2 == 0
numeros = [3, 6, 4, 8, 7, 9, 2, 5]
pares = [x for x in numeros if eh_par(x)]
print(pares)

# exemplo 2 - negativos sem função pronta
numeros = [5, -3, 1, 4, 7, -8, -2]
negativos = [x for x in numeros if x < 0]
print(negativos)
```

2.7.3. Reduce

A última função especial de alta ordem envolvendo coleções que estudaremos é o `reduce`. Além da função e da coleção, ele receberá também um valor inicial. Ele irá aplicar a função entre o valor inicial e o primeiro valor da coleção. Em seguida, entre o resultado dessa operação e o segundo valor da coleção. Depois, entre o resultado desta operação e o terceiro valor da coleção, e assim sucessivamente. Ou seja, ele **acumula** uma operação ao longo de uma coleção. O exemplo mais tradicional é o somatório.

Se você possui uma lista contendo os valores [1, 3, 5, 7, 9] e utilizar o `reduce` com valor inicial 0, ele retornará o resultado de:

(((((0 + 1) + 3) + 5) + 7) + 9)

No Python, o `reduce` não é uma função nativa como o `map` e o `filter`, e devemos importá-la de `functools`.

```
from functools import reduce

lista = [1, 3, 5, 7, 9]

somatorio = reduce(lambda x, y: x + y, lista, 0) # função: o lambda criado; coleção: lista; valor inicial: 0

print(somatorio)

# colocando valor inicial 5

somatorio_inicial = reduce(lambda x, y: x + y, lista, 5)
print(somatorio_inicial)
```

Saída na tela:

```
25
30
```

Um uso muito legal do `reduce` é para agrupar dados em categorias. Considere a estrutura abaixo, que lista os cursos de diferentes professores da Ada:

```
professores = {
    'André': 'Python',
    'Bruna': 'DevOps',
    'Cabral': 'JavaScript',
    'Rafael': 'Python',
}
```

Vamos criar uma estrutura por categoria. Ela fará nosso papel de acumulador e será algo assim:

```
{'Python': [], 'JavaScript': [], 'DevOps':[]}
```

Ela será determinada de maneira **automática**, ou seja, vamos determinando as disciplinas e adicionando ao dicionário.

Por fim, precisamos criar a função que usaremos. Para fugir de usar variáveis globais, podemos fazer uma função que gera funções a partir de um dicionário, que tal?

```
from functools import reduce

def gera_redutor(dicionario):

    def redutor(acumulador, chave):
        if dicionario[chave] in acumulador:
            acumulador[dicionario[chave]].append(chave)
        else:
            acumulador[dicionario[chave]] = [chave]
        return acumulador

    return redutor

professores = {
    'André': 'Python',
    'Bruna': 'DevOps',
    'Cabral': 'JavaScript',
    'Rafael': 'Python',
}

redutor_profs = gera_redutor(professores)

profs_por_curso = reduce(redutor_profs,
    professores,
    {})

print(profs_por_curso)
```

Saída na tela:

```
{'Python': ['André', 'Rafael'], 'DevOps': ['Bruna'], 'JavaScript': ['Cabral']}
```

A ideia não é que você nunca mais utilize técnicas de programação imperativa. O Python não é uma linguagem funcional pura, e muitos de seus recursos mais úteis estão ligados à programação orientada a objeto, por exemplo.

Porém, você agora é capaz de analisar de maneira mais crítica o impacto de diferentes técnicas para resolver um problema e fazer uma escolha mais consciente refletindo sobre os seus benefícios e suas desvantagens.

As ferramentas funcionais são mais instrumentos para sua caixinha de ferramentas, que podem ser incorporados parcialmente em programas orientados a objeto, como ocorre com frequência, por exemplo, no desenvolvimento web em JavaScript.

Algumas referências (em inglês) caso você queira se aprofundar:

Uma explicação mais profunda sobre como recursão funciona na memória do computador e como isso muda quando implementamos recursão de cauda: https://towardsdatascience.com/python-stack-frames-and-tail-call-optimization-4d0ea55b0542

Um pouco mais sobre recursão vs loops: https://arstechnica.com/information-technology/2013/04/recursion-or-while-loops-which-is-better

O Real Python possui alguns tutoriais e artigos breves e gratuitos aqui de programação funcional em Python e um curso (pago) mais completo aqui. O material completo deles inclui mais detalhes também sobre map, filter e reduce. O índice está disponível aqui.