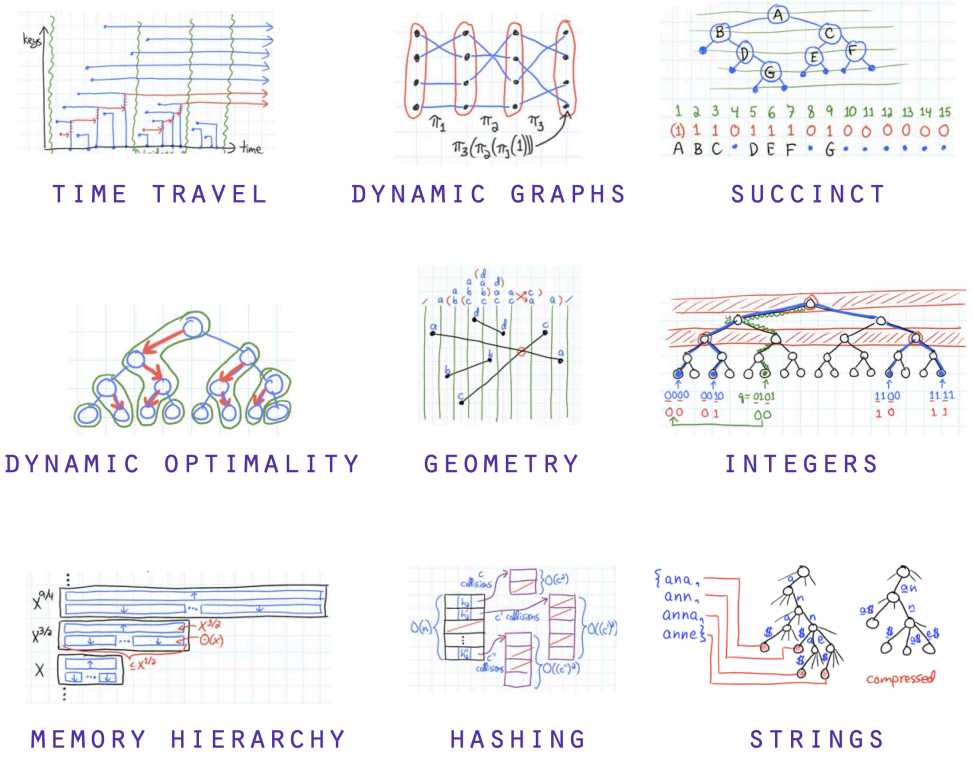
Estrutura de Dados

Listas e intro a programação funcional

Prof. Saulo Oliveira

Análise e Des. de Sistemas



Funções

Funções

> ⚠️ Um problema complexo pode ser simplificado quando dividido em vários problemas menores que são mais fáceis de resolver.

- Redução de complexidade;
- Permite focalizar a atenção em um problema pequeno de cada vez;
- Produz melhor compreensão do todo.

Analogia com o corpo humano:

- Corpo humano 🧍 ≺ Sistemas 👱 ≺ Órgãos 🫀 ≺ Células ⚛️ ≺ Móleculas 🧬.
- Módulos ≺ Bibliotecas ≺ Funções ≺ Variáveis.

Funções

⚠️ As funções são blocos de instruções que realizam tarefas específicas. **O código de uma função é carregado uma vez e pode ser executado quantas vezes forem necessárias.** As funções permitem a realização de desvios na execução.

```
def porteiro(nome):

    qtd_letras = len(nome)

    if qtd_letras < 6:
        comprimento = 'curto'
    else:
        comprimento = 'comprido'

    print(f'Olá, {nome}!')
    print(f'Seu nome é {comprimento}.')
    print('Boa continuação do de dia!')
```

```
porteiro('Saulo Oliveira')

s = input('Digite seu nome: ') #
porteiro(s)

porteiro(s[:5][::-1])
```

Funções

Vejamos o exemplo da função `len` no Python.

Definição: A função `len` retorna o comprimento (o número de itens) de um objeto. O argumento pode ser uma sequência (como uma `string`, `bytes`, `tuple`, `list` ou `range`) ou uma coleção (como um `dict`, `set` ou `frozenset`).

```
nome = input('Digite seu nome')
l = len(nome)
print(l)
```

```
# bultins.py -- fake
def len(s: Sized) -> int:
    contador = 0

    for _ in s:
        contador += 1

    return contador
```

Anatomia de uma função

- Na **definição** de funções, as variáveis recebem um nome e são chamadas de **parâmetros**;
- Funções processam algo e devolvem/retornam valores após sua execução. Para isto, utiliza-se o comando `return` e valores são retornados.
- Para capturar os valores retornados, as funções devem aparecer do lado direito de uma expressão de atribuição;
- Usamos *dicas* para os sinalizarmos os tipos de valore. Usamos duas sintaxes, a saber, `: int` para variáveis e `-> int` para funções.

```python
#somatorio.py

def somatorio(n: int) -> int:
    total = 0

    for i in range(n + 1):
        total += i

    return total

s1 = somatorio(10)
s2 = somatorio(4)
s3 = somatorio(-3)

print(f'A soma até 10 é {s1}.')
print(f'A soma até 4  é {s2}.')
print(f'A soma até -3 é {s3}.')
```

Escopo de variáveis

⚠️ **Escopo** refere-se à abrangência em que uma variável estará disponível no seu programa. Na maioria das linguagens de programação há dois escopos, a saber, o `global` e o `local`.

- As variáveis que são declaradas com o **escopo global** estão disponíveis em qualquer região de seu programa, independentemente do tamanho que seu programa possua;
- Já as variáveis de **escopo local** estão disponíveis, apenas, na região em que foram declaradas.

Por exemplo, uma variável que foi definida dentro de uma função existe apenas dentro daquela função. **Após a execução de uma função, o valor dessa variável não mais existirá e, se não for armazenado em uma variável global, ele será descartado.**

Passagem de parâmetro por valor

- Parâmetro da função se comporta como uma variável local e qualquer alteração no parâmetro só é perceptível dentro da função;
- Depois que a função é finalizada, a variável que foi passada como parâmetro por valor contém o valor que tinha no momento da chamada da função.

```
numero = int(input("Digite um numero")) # 11

def dobro(numero):
    numero = numero * 2
    return numero

dobrado = dobro(numero)

print(dobrado)   # imprime quanto?
print(numero)    # imprime quanto?
```

Passagem por valor e passagem por referência

Fonte: https://devblog.drall.com.br/excelente-imagem-animada-gif-para-ensinarrepresentar-passagem-de-parametros-por-valorcopia-e-por-referencia.

EXERCÍCIOS

Exercícios (1)

1. Faça uma função que aceite dos parâmetros e calcule as quatro operações básicas e retorne esses quatro valores de uma só vez.
2. Indique os itens que são verdadeiros sobre funções em Python:

- Uma função pode retornar apenas um único valor.
- Uma função pode retornar vários valores.
- A função nunca retorna nada a menos que você adicione uma instrução de `return`.

3. Qual a saída do código abaixo?

```
def fun1(num):
    return num + 25
fun1(5)
print(num)
```

Exercícios (2)

4. Faça uma função que recebe a média final de um aluno por parâmetro e retorna o seu conceito, conforme a tabela ao lado.
5. Faça uma função que recebe, por parâmetro, uma lista de 05 inteiros e substitui todos os valores negativos dela por zero.
6. Faça uma função que recebe um valor inteiro e verifica se o valor é positivo ou negativo. A função deve retornar um valor booleano (`True` ou `False`).
7. Escreva uma função que conte quantos números ímpares existe em uma lista e retorne uma tupla com essa indicação.

Nota	Conceito
de 0,0 a 4,9	D
de 5,0 a 6,9	C
de 7,0 a 8,9	B
de 9,0 a 10,0	A

Recursão

Recursão (1)

⚠️ Recurso elegante em que uma função chama a si mesma. Quando isso ocorre, dizemos que a função é recursiva. **A recursão precisa de um caso base.**

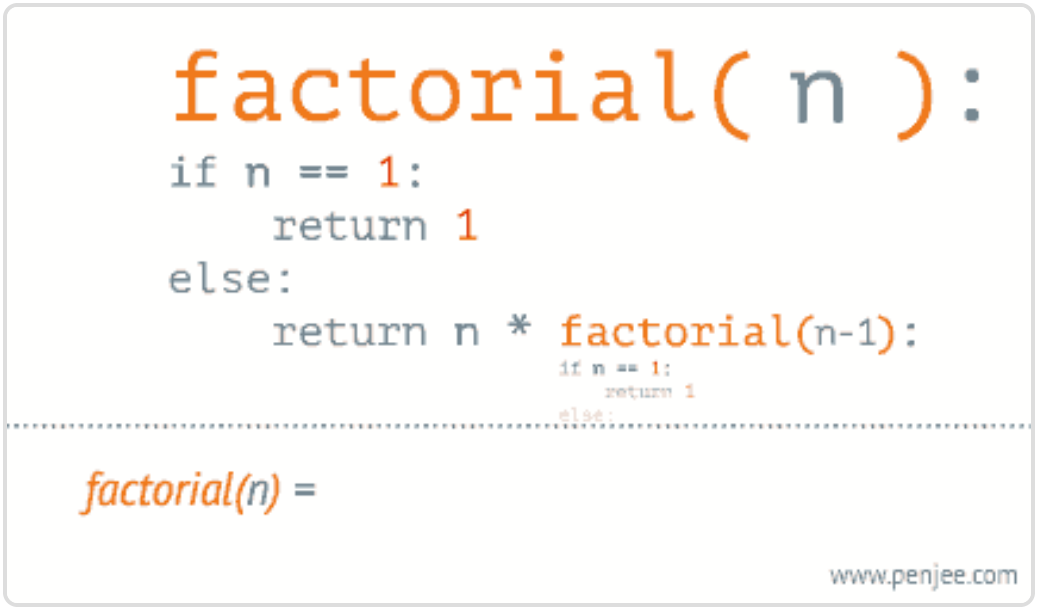
Vejamos o problema do fatorial:

```python
def factorial(n):
    fat = 1

    for i in range(1, n + 1):
        fat = fat * i
    return fat

n = int(input("Digite o n: "))

print(fatorial(n))
```



Recursão (2)

A sequência de Fibonacci é outro problema clássico no qual podemos aplicar recursão.

- A sequência começa com dois números, a saber, 1 e 1;
- Os números seguintes são a soma dos dois números anteriores.

$$1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

```
def fibonacci(n):
    f1 = 1
    f2 = 1
    if n > 2:
        for i in range(2, n):
            f1, f2 = f2, f1 + f2
    return f2
n = int(input("Digite o n: "))
print(fibonacci(n))
```

```
def fibo(n):
    if n > 2:
        return fibo(n-1) + fibo(n-2)
    else:
        return 1
n = int(input("Digite o n: "))
print(fibo(n))
```

Exercícios

Observação importante: tem escrito *Escreva um programa*, mas é pra fazer os *scripts* com **funções recursivas.**

1. Escreva um programa para imprimir os primeiros 50 números naturais usando recursão.
2. Escreva um programa para calcular a soma dos números de 1 a n usando recursão.
3. Escreva um programa para imprimir os elementos da lista (informada pelo usuário) usando recursão.
4. Escreva um programa para converter um número decimal em binário usando recursão.
5. Escreva um programa que realiza busca binária usando recursão.

PROGRAMAÇÃO FUNCIONAL

Programação funcional

- A programação funcional é uma **abordagem mais abstrata**;
- Programa visto como **avaliações de funções** matemáticas/lógicas;
- **Mais focado no quê computar do que em como computar**. Isso amplia sua perspectiva em programação;
- **Programação funcional pura é difícil**;
- Linguagens pegam emprestados conceitos do mundo funcional. **Python não é funcional**, mas dá suporte para algumas técnicas.

Funções, cidadãs de primeira ordem

As funções em Python podem ser usadas como dados de entrada (parâmetros) apenas referindo-se a elas por nome (sem argumentos) ou criando uma `lambda` (função anônima);

É importante respeitar alguns princípios deste paradigma quanto ao uso de funções:

- As **funções** precisam ser **puras** 😏;
- Deve-se adotar **composição de funções** 🤝;
- Deve-se evitar a qualquer custo os **efeitos colaterais** 😷.

Lambdas (Closures, blocks, etc)

Vejamos a função que eleva um dado valor ao quadrado e soma 1:

```
# quad.py
def quad_mais_um(x):

    quad = x ** 2
    quad += 1

    return quad
```

Uma definição como a acima é equivalente a:

```
quad_mais_um = lambda x : x ** 2 + 1
```

Infelizmente não lambdas com mais de uma linha (Ruby pode!).

Programação funcional

> ⚠️ As funções do Python podem ser usadas como dados de entrada (parâmetros) apenas referindo-se a elas por nome (sem argumentos) ou criando uma expressão lambda.

Qual é o uso disso?

- Em Python, pode definir funções que levam outras funções como seus argumentos;
- Estas são conhecidas como funções de **ordem superior**;
- Várias delas já são nativas (`map` , `filter` e `reduce` *).

*A função `reduce` não é nativa, tem que importar do pacote `functools` .

Map

⚠️ Função que nos permite converter uma lista em outra lista (relacionada) do mesmo tamanho, em que os elementos da segunda lista são funções dos elementos da primeira lista.

```
dobro = lambda x : 2 * x

lista_origem = [1, 2, 3, 4]
lista_resultado = []

for e in lista_origem:
    result = dobro(e)
    lista_resultado.append(result)

print(lista_resultado) #???
```

Map

⚠️ Função que nos permite converter uma lista em outra lista (relacionada) do mesmo tamanho, em que os elementos da segunda lista são funções dos elementos da primeira lista.

```
dobro = lambda x : 2 * x

lista_origem = [1, 2, 3, 4]
resultado = []

for e in lista_origem:
    result = dobro(e)
    resultado.append(result)

print(resultado) #???
```

```
dobro = lambda x : 2 * x

origem = [1, 2, 3, 4]

resultado = list(map(dobro, origem))

print(resultado) #???
```

⚠️ A função `map` não muda a lista original.

Filter

⚠️ Às vezes, queremos selecionar certos elementos de uma lista que satisfaçam determinadas propriedades; As propriedades podem ser representadas por um predicado – uma função que retorna um valor booleano, i.e., `True` ou `False`.

```
lista_origem = [-1, 2, -3, -4]
lista_resultado = []

for elemento in lista_origem:
    if elemento >= 0:
        lista_resultado.append(elemento)

print(lista_resultado) #???
```

Filter

Existe uma função de ordem superior chamada `filter` que recebe dois argumentos:

a) uma função (um predicado, ou seja, retornar um booleano); e

b) uma lista.

A função `filter` retorna uma nova lista que consiste em todos os elementos da lista original que satisfizeram o predicado (para o qual o predicado retornou `True`);

A lista original é *filtrada* para fornecer a nova lista.

Filter

```
def positivo(x):
    return x >= 0

a = list(filter(positivo, [-3, 1, -4, 1, -5, 9, -2, 6]))

b = list(filter(lambda x: x > 0, [5, -3, -8, 9, 7, -9]))

c = list(filter(lambda x: x != 0, [1, 0, 0, 2, 0, 0, 0]))

d = list(filter(lambda x: x > 5, [4, 1, -2, 0, 3]))

e = list(filter(lambda x: x > 10, []))

print(a, b, c, d, e, sep='\n')
```

Mapeando e Filtrando

```
def modulo(n):
    return n if n > 0 else n * -1

par = lambda x : x % 2 == 0

valores = [-3, 1, -4, 1, -5, 9, -2, 6]

positivos = map(modulo, valores)
pares_positivos = filter(par, positivos)
elementos = list(pares_positivos)

print(elementos)
```

Reduce (1)

⚠️ Às vezes, queremos agregar os elementos de uma lista e torná-los um único valor. A função `reduce` retorna um único valor com base numa operação/função de redução. Essa operação de redução precisa **exatamente de dois valores e transforma-os em um**.

Por exemplo:

- somar todos os elementos de uma lista juntos;
- multiplicar todos os elementos de uma lista juntos;
- encontrar os maiores / menores elementos de uma lista;

Em todos os casos, estamos **reduzindo** uma lista a um único valor.

Reduce (2)

Exemplo da soma de uma lista:

Se os elementos da lista são `[i, j, k, l ...]`, queremos computar `i + j + k + l`, ou seja:

1. `(i + j)`
2. `((i + j) + k)`
3. `(((i + j) + k) + l)`

até que todos os elementos da lista sejam adicionados juntos. Em cada caso, estamos adicionando a soma anterior ao próximo elemento. Esse é o padrão da redução.

Para usarmos a função `reduce` precisamos importá-la do módulo `functools`.

Reduce (3)

```
from functools import reduce
lista = [1, 2, 3, 4, 5]
redutor = lambda x, y: x + y

soma = reduce(redutor, lista)
print(f'A soma de {lista} é {soma}')
```

O código acima avalia a lista da seguinte forma:

`((((1 + 2) + 3) + 4) + 5)`

que é a mesma coisa que somar a lista toda.

Assim como as outras funções de ordem superior `map` e `filter`, `reduce` evita a gente ter de usar estruturas de repetição de forma explícita.

Revisão

Revisão

Para encerrar, as funções de alta ordem `map`, `filter` e `reduce` explicadas com emojis:

```
from functools import reduce
from ifce.ads import cozinhar, is_vegano, comer

base =  [🐮, 🥔, 🐔, 🌽, 🦐]

cozidos = map(cozinhar, base)
#cozidos = [🍔, 🍟, 🍗, 🍿, 🍤]

veganos = filter(is_vegano, cozidos)
# veganos = [🍟, 🍿]

resultado = reduce(comer, cozidos)
# resultado = 💩
```

EXERCÍCIOS

Exercícios

1. Escreva um programa para calcular a soma dos números de 1 a n sem usar recursão e sem usar laços.
2. Escreva um programa para calcular o fatorial até n (informado pelo usuário) sem usar recursão e sem usar laços.
3. Escreva um programa que conte quantos números ímpares existe em uma lista sem usar recursão e sem usar laços.
4. Escreva um programa que retorne o máximo valor de uma lista sem usar recursão e sem usar laços.