Funções e Dicionários em Python

Modulo 01 - Programação Orientada a Objetos

ANTÔNIO DAVID VINISKI
antonio.david@pucpr.br
PUCPR

Agenda

- Apresentação
- Programação Estruturada – Procedural
- Funções
  - Parâmetros posicionais
  - Parâmetros nomeados
  - Funções com número variado de parâmetros
  - Funções com retorno de dados
  - Função Lambda
  - Documentação de Função
  - Anotação de funções
- Dicionários

Quem sou eu?

Who am I?

APRESENTAÇÃO

Quem sou eu?

- Engenheiro de Computação – UEPG (2011 - 2015)
- Mestre em Computação Aplicada – UEPG (2016 - 2018)
- Professor Substituto – IFPR Telêmaco Borba (2018 - 2019)
- Doutor em Informática – PUCPR (2019 - 2023)
- Cientista de Dados – HIMARKET (2019 - 2022)
- Professor Auxiliar – PUCPR – (2022 - atual)

CONTATOS

Quem sou eu?

- ANTÔNIO DAVID VINISKI
  - EMAIL: antonio.david@pucpr.br
  - LINKEDIN - https://www.linkedin.com/in/antonio-david-viniski-a18812b8/

Programação Estruturada

- Na programação estruturada:
  - Procedimentos são implementados em blocos e a comunicação entre eles se dá pela passagem de dados.
  - Um programa estruturado, quando em execução, é caracterizado pelo acionamento de procedimentos cuja tarefa é a manipulação de dados.

Programação Estruturada - exemplo

- Considere um programa para um banco financeiro.
  - É fácil perceber que uma entidade importante para o nosso sistema será uma **conta**.
- Primeiramente, suponha que você tem uma conta nesse banco com as seguintes características: **titular, número, saldo e limite**

Programação Estruturada - exemplo

Como você armazenaria os dados de uma conta corrente considerando o que você aprendeu até o momento em Python?

- Podemos começar inicializando essas características

```
print("Essa é a conta 1")
numero = "1023780-9"
titular = "Antônio"
saldo = 1500.00
limite = 2000.00
```

Programação Estruturada - exemplo

- E se precisássemos criar mais uma conta?

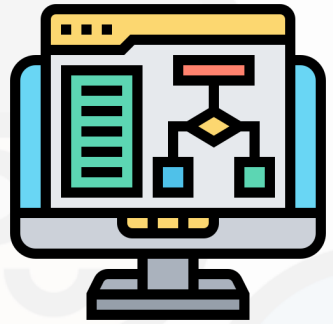
```
print("Essa é a conta 1")
numero1 = "1023780-9"
titular1 = "Antônio"
saldo1 = 1500.00
limite1 = 2000.00

print("Essa é a conta 2")
numero2 = "1041234-2"
titular2 = "Maria"
saldo2 = 2700.00
limite2 = 2200.00
```

- Dessa forma fica muito difícil gerenciar as múltiplas contas que essa aplicação financeira pode ter, podemos então utilizar funções e listas para armazenar os dados

Armazenamento de Múltiplas contas

- Considerando que podemos ter várias contas armazenadas, fica difícil estruturar a nossa aplicação para armazenar todos os dados, de todas elas em variáveis.
- Para isso, o Python possui alguns tipos especiais que auxiliam na manipulação de dados que se repetem ao longo do ciclo de vida da aplicação

Exercício 1



- Crie um programa que receba os dados da conta do usuário: **titular, número, saldo e limite** e imprima a seguinte mensagem:

Conta: ‘numero’; Titular ‘nome titular’; Saldo: ‘valor saldo’; Limite de ’valor limite’

https://docs.python.org/pt-br/3/tutorial/functions.html

Funções

- A palavra reservada **def** inicia a definição de uma função.
- Ela deve ser seguida do **nome da função** e da lista de parâmetros formais entre parênteses.
- Os comandos que formam o corpo da função começam na linha seguinte e devem ser indentados.
- Opcionalmente, a primeira linha do **corpo da função** pode ser uma literal *string*, cujo propósito é documentar a função.

Funções II

- Funções são blocos de código que executam funcionalidades específicas.
- Normalmente são utilizados para evitar que determinada parte do seu código seja escrito várias vezes.

```
def funcao():
    print("Bloco de código")
```

- Para "chamar" uma função, utilizamos o nome que foi definido, dessa forma:

```
funcao()
```

Funções - parâmetros

- Além de executar um bloco de código, funções também podem receber parâmetros e retornar dados.
- O envio de dados para um função ocorre por meio dos parâmetros que ela pode receber.

```
def imprimir(nome):
    print(f"Nome: {nome}")

imprimir("João da Silva")
imprimir("José de Jesus")
imprimir("Maria das Dores")
```

- Caso nenhum valor seja informado ao chamar a função, um erro será gerado, pois o parâmetro é obrigatório neste exemplo.

Funções – parâmetros com valores padrão

- Podemos resolver o problema da obrigatoriedade da passagem de parâmetros definindo um **valor padrão** para os parâmetros de uma função.
- A utilização dos **valores padrão** serve para dar um valor quando quem chamou a função não passar nenhum valor para os parâmetros definidos.

```
def horarios(disciplina='Programação em Python', horario='13:00 as 17:00'):
    print(f"A horário da disciplina {disciplina} é {horario}")

horarios()
horarios("Desenvolvimento Ágil", "08:00 as 12:00")
```

- Caso nenhum valor seja informado ao chamar a função, o valor padrão dos parâmetros é utilizado.

Chamada de Função

- Em uma chamada de função, argumentos nomeados devem vir depois dos argumentos posicionais.
- Todos os argumentos nomeados passados devem corresponder com argumentos aceitos pela função.

Chamada de Função Posicional

- Quando chamamos uma função, podemos utilizar a localização (ordem) dos parâmetros para fazer o casamento entre o que foi informado e o que a função espera receber.

```
def matricula(curso='', disciplina='', periodo=0):
    print(f'Realizando a matrícula: \n'
          f'\t-Curso: {curso}\n'
          f'\t-Disciplina: {disciplina}\n'
          f'\t-Período: {periodo}')
matricula("Ciência da Computação", "RA", 2)
```

- Essa é uma chamada de **função posicional**, ou seja: que respeita a ordem dos parâmetros.
- Outra forma de fazer essa chamada de função é utilizar os nomes dos parâmetros!

Chamada de Função Nomeada

- Não é necessário seguir a ordem dos parâmetros em um chamada de função nomeada.

```
def matricula(curso='', disciplina='', periodo=0):
    print(f'Realizando a matrícula: \n'
          f'\t-Curso: {curso}\n'
          f'\t-Disciplina: {disciplina}\n'
          f'\t-Período: {periodo}')

matricula("Ciência da Computação", "RA", 2)
matricula(periodo=2,  curso="Ciência da Computação",  disciplina="RA")
```

- A saída das duas chamadas da função será a mesma, pois utilizamos os nomes dos parâmetros. O Python saberá qual valor referencia qual parâmetro.

Número variado de parâmetros posicionais

- Caso você queira desenvolver uma função que recebe um número variável de parâmetros, você pode utilizar o parâmetro ***args**.
- Esses argumentos serão empacotados em uma tupla e você poderá processá-los com um loop **for**, por exemplo.

```
def maior_30(*args):
    print(args)
    for num in args:
        if num > 30:
            print(num)

maior_30(10, 20, 30, 40, 50, 60)
```

O nome ***args** é uma convenção, ou seja uma boa prática entre programadores Python! Contudo, podemos alterar esse nome por outro que faça mais sentido para a aplicação

Número variado de parâmetros nomeados

- Se quisermos desenvolver uma função com número variado de parâmetros nomeados, devemos utilizar o parâmetros **kwargs.
- Dessa forma, todos os dados passados à função serão guardados nessa variável **kwargs, em formato de um dicionário.
- Podemos então percorrer os itens do dicionário para recuperar o nome (chave) e o valor de cada um dos parâmetros nomeados.

O nome **kwargs também é uma convenção, com isso podemos alterá-lo se necessário

```
def dadosPessoais(**kwargs):
    print(type(kwargs))
    for chave, valor in kwargs.items():
        print(f"{chave}: {valor}")

dadosPessoais(nome='João', idade=35, carreira='Professor Auxiliar')
```

Funções com retorno de dados

- As funções também podem retornar valores através da palavra reservada **return**.

```
def operacao(val1, val2, operador):
    match(operador):
        case "+": return val1 + val2
        case "-": return val1 - val2
        case "*": return val1 * val2
        case "/": return val1 / val2
        case "_": return "Operador inválido"

valor = operacao(1,45,"/")
print(valor)
```

Funções com retorno múltiplos

- Funções também podem retornar múltiplos dados, desta forma, deveremos ter múltiplas variáveis para receber o retorno da função, ou caso tenha apenas uma, essa variável será do tipo **tupla**.

```
def somaMedia(valor1, valor2):
    soma = valor1 + valor2
    media = (valor1 + valor2)/2
    return soma, media

valores = somaMedia(12, 42)
print(valores)
valorSoma, valorMedia = somaMedia(12, 42)
print(valorSoma, valorMedia)
```

Expressões lambda

- Uma função lambda é criada usando a palavra-chave lambda, seguida de um ou mais argumentos, e uma expressão:
  - argumentos são os dados de entrada que esta função irá receber
  - expressão é o código que será executado quando a função lambda for chamada.

```
lambda {argumentos}: {expressão}
```

- Função que retorna a soma de seus dois argumentos: **lambda a, b: a+b**.
- As funções lambda podem ser usadas sempre que objetos função forem necessários. Eles são sintaticamente restritos a uma única expressão.
- Como definições de funções aninhadas, as funções lambda podem referenciar variáveis contidas no escopo.

Strings de Documentoção

- Definidas no início da função com três aspas, a string de documentação é utilizada para definir o que aquela função realiza, bem como quais são seu parâmetros, retorno, etc.

```
def myFunction():
    """ Do nothing, but document it.
    No, really, it doesn't do anything."""
    pass

print(myFunction.__doc__)
```

Anotações de função

- Anotações de função são informações de metadados completamente opcionais sobre os tipos usados pelas funções definidas pelo usuário
- Anotações são armazenadas no atributo **__annotations__** da função como um dicionário e não tem nenhum efeito em qualquer outra parte da função.
- Anotações de parâmetro são definidas por dois-pontos (“**:**”) após o nome do parâmetro, seguida por uma expressão que quando avaliada determina o valor da anotação.
- Anotações do tipo do retorno são definidas por um literal **->**, seguida por uma expressão, entre a lista de parâmetro e os dois-pontos que marcam o fim da instrução **def**

Anotações de função II

- O exemplo a seguir possui um argumento obrigatório, um argumento opcional e o valor de retorno anotados

```
def f(nome: str, cep: str = '00000000') -> str:
    print("Annotations:", f.__annotations__)
    print("Argumentos:", nome, cep)
    return nome + ' e ' + cep
f('Antonio')
print(valores)
```


Dicionários
com Python

Dicionários

- Um tipo de dados útil incorporado ao Python é o dicionário.
- Ao contrário das sequências (listas), que são indexadas por um intervalo de números, os dicionários são indexados por **chaves**, que podem ser de qualquer tipo **imutável**
  - **strings** e **números** sempre podem ser chaves.

Dicionários II

- É melhor pensar em um dicionário como um conjunto de pares **chave:valor**, com a exigência de que as chaves sejam únicas (dentro de um dicionário).
  - Um par de chaves cria um dicionário vazio: **{}**.

```
conta = {}
```

Dicionários III

- As principais operações em um dicionário são armazenar um valor com alguma **chave** e extrair o valor fornecido pela chave.

```
conta = {}
conta["numero"] = "1023780-9"
conta["titular"] = "Antônio"
conta["saldo"] = 1500.00
conta["limite"] = 2000.00

print(f"O titular da conta {conta['numero']} é {conta['titular']}")
```

Criando e inicializando Dicionários

- Utilizando o par de chaves {}, criamos um dicionário vazio, se desejamos inicializar durante a criação, existem duas formas principais
  - Definindo os pares **chave:valor** entre o par de chaves

```
conta1 = {'numero':"1023780-9", 'titular': "Antônio", 'saldo': 1500.00,'limite': 2000.00}
```

  - Utilizando a função **dict**

```
conta2 = dict(numero="1041234-2", titular="Maria", saldo=2700.00, limite=2200.00)
```

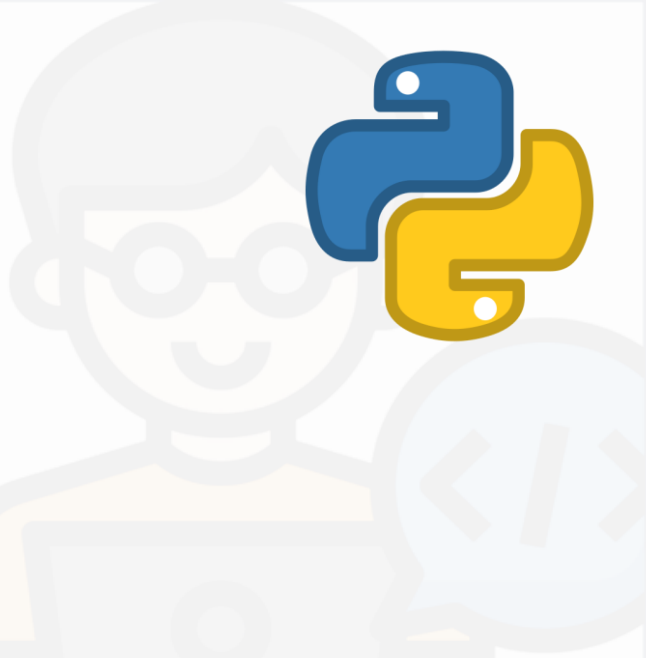
Exercício 2

- Podemos otimizar a criação de contas utilizando uma função do python, com isso:
  - Criar uma função que receba como parâmetro os dados **numero, titular, saldo, limite** e **status, e** retorne um dicionário para representar a conta.
  - Utilizar a função implementada para criar a **conta1** e **conta2**

```
conta1 = {'numero':"1023780-9", 'titular': "Antônio", 'saldo': 1500.00,'limite': 2000.00}
```

```
conta2 = dict(numero="1041234-2", titular="Maria", saldo=2700.00, limite=2200.00)
```

Funcionalidades



- E se precisássemos implementar as funcionalidades de depositar e de sacar nessa aplicação do banco financeiro, como seria possível?

Funcionalidades II

- Assim como criamos a função para criar uma conta, podemos criar uma função para realizar o depósito e outra para realizar o saque da conta.
  - No caso do deposito, devemos somar o valor de depósito ao saldo da conta.

```
conta['saldo'] += valor
```

  - Para o saque, devemos subtrair o valor do saque do saldo da conta

```
conta['saldo'] -= valor
```

  - Agora podemos criar a função depositar que recebe uma conta e o valor de depósito, bem como a função sacar, que também recebe o a conta e o valor do saque

Funcionalidades III

```
def depositar(conta, valor):
    conta['saldo'] += valor

def sacar(conta, valor):
    conta['saldo'] -= valor

depositar(conta1, 200)
print(conta1['saldo'])
depositar(conta2, 600)
print(conta2['saldo'])
sacar(conta1, 1000)
print(conta1['saldo'])
```

Exercício 3

- Na aplicação do banco, também precisamos realizar a transferência entre contas, para isso, crie um função que receba como parâmetro três argumentos:
  - transfere - conta que irá transferir
  - recebe - conta que irá receber a transferência
  - valor - o valor da transferência
- Lembre-se que na realização da transferência o saldo de uma conta (transfere) diminui e da outra (recebe) aumenta.

Exercício 3 - resolução

```
def transferir(transfere, recebe, valor):
    transfere['saldo'] -= valor
    recebe['saldo'] += valor

transferir(conta2,conta1, 600)
print(f"O saldo da conta {conta1['numero']} é {conta1['saldo']}")
print(f"O saldo da conta {conta2['numero']} é {conta2['saldo']}")
```

Exercício 4

- Para não precisar reescrever o código para imprimir o saldo da conta, crie uma função chamada **extrato** que recebe uma conta e imprime as seguinte mensagem:
  - O saldo da conta “numero” é de “saldo”!

Reflexão

- Por mais que tenhamos agrupado os dados de uma conta utilizando o dicionário, essa ligação é frágil no mundo procedural e se mostra limitada.
  - Precisamos pensar sobre o que escrevemos para não errar.
  - Podemos modificar o saldo, bem como os demais dados das contas diretamente.
  - ...
- O **paradigma orientado a objetos, que veremos a seguir,** vem para sanar essa e outras fragilidades do paradigma procedural que veremos a seguir.