Funções

Funções são blocos de código identificados por um nome, que podem receber parâmetros pré-determinados.

No Python, as funções:

- Podem retornar ou não objetos.
- Aceitam ***Doc Strings***.
- Aceitam parâmetros opcionais (com ***defaults***). Se não for passado o parâmetro será igual ao *default* definido na função.
- Aceitam que os parâmetros sejam passados com nome. Neste caso, a ordem em que os parâmetros foram passados não importa.
- Tem ***namespace*** próprio (escopo local), e por isso podem ofuscar definições de escopo global.
- Podem ter suas propriedades alteradas (geralmente por decoradores).

Documentação

PyDOC é a ferramenta de documentação do Python. Ela pode ser utilizada tanto para acessar a documentação dos módulos que acompanham o Python, quanto a documentação dos módulos de terceiros.

```
#para exibir a documentação de "modulo.py" no diretório atual
pydoc3 ./modulo.py

# a documentação pode ser vista através do browser
# http://localhost:8000/
pydoc3 -p 8000

#consultar a documentação no própior interpretador
help(list)
```

Doc Strings

Doc Strings são *strings* que estão associadas a uma estrutura do Python. Nas funções, as *Doc Strings* são colocadas dentro do corpo da função, geralmente no começo. O objetivo das *Doc Strings* é servir de documentação para aquela estrutura.

Uma **docstring** é uma string de documentação em Python que serve para documentar módulos, classes, funções ou métodos. Ela é inserida logo após a definição do objeto (módulo, classe, função ou método) e é usada para descrever o propósito e o comportamento do objeto. As docstrings são importantes para facilitar a compreensão do código, fornecendo informações sobre como usar e entender as partes do código.

Em termos de sintaxe, uma **docstring** é simplesmente uma string colocada entre aspas triplas (`'''` ou `"""`).

Função

Retornando um valor

```
def minha_funcao(parametro):
    """
    Esta é uma docstring.

    Ela descreve o propósito e o comportamento da função minh
    """
    # Corpo da função
    return parametro * 2
```

```
def minha_funcao(parametro):
    """
    Esta é a docstring da função minha_funcao.

    Ela descreve o propósito e o comportamento da função.
    """
    # Corpo da função
    return parametro * 2

# Acessando a docstring da função minha_funcao
print(minha_funcao.__doc__)
```

Os parâmetros com ***default*** devem ficar após os que não tem ***default***.

```
def func(param1, param2=padrao):
    """Doc String"""
    <bloco de codigo>
    return valor
```

```
#Para um parâmetro ser opcional, o mesmo é atribuído a um
#valor padrão (default) - o mais comum é utilizar None
```

Nomeando Parâmetros

Quando especificamos o nome dos parâmetros, podemos passá-los em qualquer ordem. Quando especificamos o nome de um parâmetro, somos obrigados a especificar o nome de todos os outros parâmetros também.

```
def retangulo(largura, altura, caractere="*"):
    linha = caractere*largura
    for i in range(altura):
        print(linha)

retangulo(3,4)
retangulo(largura=3, altura=4)
retangulo(altura=4, largura=3)
retangulo(caractere="-", altura=4, largura=3)
```

Funções como parâmetro

Um poderoso recurso de Python é permitir a passagem de funções como parâmetro. Isso permite combinar várias funções para realizar uma tarefa.

```
def soma(a,b):
    return a+b
def subtracao(a,b):
    return a-b
def imprime(a,b,foper):
    print(foper(a,b))
imprime(5,4,soma)
imprime(10,1,subtracao)
```

Funções Lambda

Podemos criar funções simples, sem nome, chamadas de funções lambda. Funções lambda são utilizadas quando o código da função é muito simples ou utilizado poucas vezes.

```
a = lambda x: x*2
print(a(3))

x= lambda a,b:(a*b/100)
x(100,5)
```

Função Recursiva

```
# algoritmo fatorial sem recursão
def fatorial(n):

    n = n if n>1 else 1
    j = 1
    for i in range(1,n+1):
        j=j*i
    return j

for i in range(1,6):
    print(i, '->', fatorial(i))
```

```
def fatorial(num):
    if num <= 1:
        return 1
    else:
        return(num*fatorial(num-1))
```

```
print(fatorial(5))
```

Retornando Múltiplos Valores

Apesar de uma função executar apenas um retorno, em Python podemos retornar mais de um valor.

```
def calculadora(x,y):
    return x+y,x-y

print(calculadora(1,2))


def calculadora(x,y):
    return {'soma': x+y, 'subtração':x-y}

resultados = calculadora(1,2)
for key in resultados:
    print(f'{key}: {resultado[key]}')
```

Número Arbitrário de Parâmetros

Podemos passar um número arbitrário de parâmetros em uma função. Utilizamos as chamadas variáveis mágicas do Python: ***args** e ****kwargs.** Não é necessário utilizar exatamente estes nomes: **args e ***kwargs . Apenas o asterisco(**), ou dois deles(**), serão necessários.

```
def teste(arg, *args):
    print(f'primeiro argumento normal: {}')
    for arg in args:
```

```
        print(f'outro argumento: {}')

teste('python', 'é', 'muito','legal')
lista = ["é", "muito", "legal"]
teste('python', *lista)
tupla=("é", "muito", "legal")
teste('python',*tupla)
```

O `*args` então é utilizado quando não sabemos de antemão quantos argumentos queremos passar para uma função. O asterisco (`*`) executa um empacotamento dos dados para facilitar a passagem de parâmetros, e a função que recebe este tipo de parâmetro é capaz de fazer o desempacotamento.

O `**kwargs` permite que passemos o tamanho variável da palavra-chave dos argumentos para uma função. Você deve usar o `**kwargs` se quiser manipular argumentos nomeados em uma função. A diferença é que o *args espera uma tupla de argumentos posicionais, enquanto o `**kwargs` um dicionário com argumentos nomeados.

```
def minha_funcao(**kwargs):
    for key, value in kwargs.items():
        print(f'{key} = {value}')

dict={'nome':'Joao', 'idade':25}
```

Módulos

Em Python, um "módulo" é um arquivo contendo definições e instruções Python. O nome do arquivo é o nome do módulo com a extensão `.py`. Um módulo pode definir funções, classes e variáveis, e também pode incluir código executável.

Aqui estão algumas características importantes dos módulos em Python:

1. **Organização de código**: Os módulos ajudam a organizar o código em unidades lógicas e separadas. Você pode agrupar funcionalidades relacionadas em um módulo e usá-lo em outros programas.
2. **Reutilização de código**: Os módulos permitem reutilizar o código em diferentes partes de um programa ou em diferentes programas. Isso reduz a redundância e facilita a manutenção do código.
3. **Encapsulamento**: Os módulos fornecem um escopo de namespace separado, o que significa que as funções, classes e variáveis definidas em um módulo não interferem com outras partes do código. Isso ajuda a evitar conflitos de nomes e facilita a identificação de onde cada parte do código está definida.
4. **Importação**: Para usar um módulo em um programa Python, você precisa importá-lo. Isso é feito usando a palavra-chave `import`, seguida pelo nome do módulo. Por exemplo, para importar um módulo chamado `meu_modulo`, você usaria `import meu_modulo`.
5. **Pacotes**: Além dos módulos individuais, você pode organizar vários módulos relacionados em um "pacote". Um pacote é simplesmente um diretório que contém um arquivo especial chamado `__init__.py` e outros módulos ou subpacotes. Isso permite uma estrutura de organização mais hierárquica para seu código.

```
# meu_modulo.py

def saudacao(nome):
    print(f"Olá, {nome}!")
```

```
# main.py
import meu_modulo

# Chamando a função saudacao do módulo meu_modulo
meu_modulo.saudacao("João")
```

Pacotes

Pacotes (***packages***) são pastas que são identificadas pelo interpretador pela presença de um arquivo com o nome "`__init__.py`". Os pacotes funcionam como coleções para organizar módulos de forma hierárquica.

O arquivo "`__init__.py`" pode estar vazio ou conter código de inicialização do pacote ou definir uma variável chamada *__all__*, uma lista de módulos do pacote serão importados quando for usado "*". Sem o arquivo, o Python não identifica a pasta como um pacote válido.

Suponha que temos um pacote construído da seguinte forma:

```
meu_pacote/
    __init__.py
    modulo1.py
    modulo2.py
```

```
# __init__.py

print("Inicializando o pacote meu_pacote")

# Definindo uma função para inicializar o pacote
def inicializar():
    print("Pacote meu_pacote inicializado")
```

```
# modulo1.py

def funcao_modulo1():
    print("Esta é uma função do módulo 1")
```

```
# modulo2.py

def funcao_modulo2():
    print("Esta é uma função do módulo 2")
```

```
# main.py
from meu_pacote import inicializar, modulo1, modulo2
# pode usar * para importar tudo

# Chamando a função para inicializar o pacote
inicializar()

# Chamando as funções dos módulos
modulo1.funcao_modulo1()
modulo2.funcao_modulo2()
```

Escopo de Variáveis

- **Escopo Local**: Variáveis definidas dentro de uma função têm um escopo local e só podem ser acessadas dentro dessa função. Se uma variável local tiver o mesmo nome que uma variável global, a variável local terá precedência dentro do escopo da função.
- **Escopo Global**: Variáveis definidas fora de qualquer função têm um escopo global e podem ser acessadas de qualquer lugar do código, incluindo dentro de funções. Para modificar uma variável global dentro de uma função, é necessário usar a palavra-chave `global`.
- **Escopo de Função Encapsulada (Enclosing)**: Em funções aninhadas, as variáveis são pesquisadas primeiro no escopo local, depois no escopo da função que envolve a função atual e, finalmente, no escopo global. Para modificar uma variável no escopo da função que envolve a função atual em uma função aninhada, é necessário usar a palavra-chave `nonlocal`.

Exercício da Aula

1. Faça um programa que controle uma agenda. Com as funções `apaga()`, `altera()`, `lista()`, `le()`, `grava()`, `menu()`. Depois faça as seguintes melhorias:
    a. Exiba o tamanho da agenda no menu principal;
    b. Exiba a posição de cada elemento;
    c. Exiba a opção de ordenar a lista por nome no menu principal;
    d. Exiba uma mensagem de erro caso duas entradas na agenda tenham o mesmo nome;
    e. Adicione também a data de aniversário e email de cada pessoa.
    f. Permita o cadastro de tipos de telefone: celular, fixo, residência ou trabalho.