A UTILIZAÇÃO DE MÉTODOS SET PARA SINTAXE E CARREGAMENTO DE FUNÇÕES EM PYTHON

THE UTILIZATION OF SET METHODS FOR SYNTAX AND LOADING OF FUNCTIONS IN PYTHON

WEICH, Rodrigo Gomes[1]

Cursando Sistemas de Informação – Faculdade Iguaçu

ABATTI, Alex Onning[2]

Cursando Sistemas de Informação – Faculdade Iguaçu

RESUMO

Este estudo tem como objetivo demonstrar o que podemos criar utilizando métodos simples da biblioteca Python, desde a criação de sintaxes e carregamento de funções personalizadas desenvolvidas tanto para desenvolvimento cotidiano como para desenvolvimentos específicos.

Palavras – chave: Métodos, Conjuntos, Estrutura de dados, Sintaxe de funções, Carregamento de funções.

ABSTRACT

This study aims to demonstrate what we can create using simple Python library methods, such as creating syntax and loading custom functions for both daily development and specific developments.

***Keywords**: Methods, Sets, Data structure, Function syntax, Function loading.*

1 INTRODUÇÃO

Módulos são partes de um sistema e são responsáveis por realizar determinadas tarefas pré-definidas, ou seja, ele realiza um determinado processo de informações definidas pelo sistema e/ou usuário, capaz de atender as necessidades do sistema ou do usuário.

Para a criação de sintaxes e carregamento de funções (pré-definidas pelo usuário), utilizaremos o sistema de Conjuntos (Sets) da linguagem Python, onde aqui, ela será composta por três partes dispostas na tabela abaixo:

Tabela 1 - Inserção de um novo comando

	Definição	Exemplo
Conjunto	Nome	tabela
Comando	Nome	soma
Sintaxe	Mensagem	soma <p> <s>
Função	Função de Execução	soma(p, s)
Na prática	tabela["comando"] = "sintaxe|função"	

Fonte: O autor

2 A DECLARAÇÃO NA PRÁTICA

```
tabela = {}

def soma(p, s):
    return print(f"O resultado da soma entre {p} e {s} é: {p+s}")

tabela["@soma"]     = "@soma <p> <s>|soma(9, 3)"

def carregar_def(funcao):
    try:
        exec(funcao)
    except:
        print(f"Função {funcao} não encontrada.")

def tratar_informações(argumentos):
    tabela_arg      = argumentos.split("|")
    sintaxe         = tabela_arg[0]
    evento          = tabela_arg[1]
    print(f"Sintaxe do comando: {sintaxe}")
    carregar_def(evento)

print(f"Comandos: {list(tabela.keys())}")

comando = input("Digite um comando: ")

if comando in tabela:
    tratar_informações(tabela.get(comando))
else:
    print(f"O comando {comando} não existe.")
```

2.1 COMO FUNCIONA

É necessário declararmos o nome do conjunto que irá receber o comando e suas atribuições, neste caso, utilizamos “tabela = {}” para identificar que este conjunto é uma tabela de comandos. Esta declaração será a base de toda a estrutura, pois é a

partir dela que as informações serão manipuladas de acordo que os métodos consigam interpretar e realizar sua execução.

Iremos declarar uma função base, que posteriormente servira para realizarmos um teste de funcionalidade, está função executará duas informações do tipo inteiro onde ela deverá efetuar sua soma e retornar ao usuário a resposta obtida.

Declaremos então:

```
def soma(p, s):
    return print(f"O resultado da soma entre {p} e {s} é: {p+s}")
```

Neste momento, já podemos criar um novo comando dentro do conjunto, tendo em mente que, é necessário seguir sempre a ordem correta de inserção e utilizar a barra vertical para separar os campos, como é mostrado no exemplo da Tabela 1.

```
tabela["@soma"]     = "@soma <p> <s>|soma(9, 3)"
```

Para executar as funções que iremos inserir dentro dos conjuntos, precisamos criar duas funções, uma será responsável por tratar as informações do conjunto e inseri-las na segunda função, que será a função responsável pelo carregamento destas informações. Na função de carregamento, utilizaremos de base a estrutura try except com o método exec().

Função de carregamento:

```
def carregar_def(funcao):
    try: #Vai executar a função, independentemente dela existir ou não
        exec(funcao) #Executa de fato a função
    except: #Caso não exista, retorna uma mensagem de erro ao usuário
        print(f"Função {funcao} não encontrada.")
```

A estrutura try except é parecida com a estrutura try catch, que é utilizada na linguagem Java. O comportamento dessa estrutura baseia-se na forma de interpretação das informações, onde o try é responsável pela execução de informações e o except, responsável pelo tratamento de possíveis erros. Em outras palavras, o try executa um bloco de comandos e, se caso ele não consiga ser executado como o esperado, ele passa para o except, que “retorna” algo informando ao usuário os erros que ocorreram durante a execução. Os erros que podem ocorrer ficam a cargo do desenvolvedor implementar um retorno ou apenas continuar a execução da aplicação.

Função de inserção:

```
def tratar_informações(argumentos):
    tabela_arg      = argumentos.split("|")
    sintaxe         = tabela_arg[0]
    evento          = tabela_arg[1]
    print(f"Sintaxe do comando: {sintaxe}")
    carregar_def(evento)
```

A função que realiza o tratamento de uma informação contém um elemento que é responsável por realizar a divisão da informação em informações menores, assim como o endereçamento de cada uma delas, este é definido como um vetor que utiliza como base o método split(). O tratamento de uma informação, nesse contexto, acontece da seguinte forma:

Tabela 2 - Divisão do argumento

Argumento inserido	“@soma <p> <s>|soma(9, 3)”	
Posição do vetor (tabela_arg)	0	1
Divisão do argumento	Sintaxe	Função
	“@soma <p> <s>”	“soma(9,3)”

Fonte: O autor

A divisão do argumento pode ser muito diversificada, porém deve-se seguir um padrão, para que não haja erro durante a interpretação das informações. Como podemos ver, argumento da Tabela 2 foi dividido e recebido por dois elementos diferentes, a sintaxe, que recebeu o valor do vetor na posição 0, e o evento, que recebeu o valor do vetor na posição 1, ambos são demonstrativos e podem ser ampliados de acordo com a estrutura que se deseja criar na hora de declarar o comando.

O elemento sintaxe, neste caso, irá servir apenas para mostrar ao usuário a sintaxe do comando em questão, já o elemento evento, será inserido na função carregar_def(), onde será executado.

Os comandos para o usuário

```
print(f"Comandos: {list(tabela.keys())}")
```

O comando acima é opcional, quando o algoritmo for executado, será mostrado ao usuário os comandos disponíveis na tabela através do método list(), onde este, irá permitir ao usuário um visual em forma de lista sem quebra de linha, retornado em

forma de mensagem da seguinte forma: Comandos: ['@soma'], podemos obter o mesmo resultado sem utilizar o método list(), porém, a mensagem retornada será em forma de dict: Comandos: dict_keys(['@soma']).

Dentro do método list() definimos o nome do conjunto onde ele deverá buscar as “chaves” e adicionar também, a expressão .keys() no final, que no caso, retornara ao usuário os comandos existentes definidos anteriormente.

Para que o usuário possa digitar um comando, basta apenas adicionarmos um “input” ao algoritmo, conforme mostrado abaixo:

```
comando = input("Digite um comando: ")
if comando in tabela:
    tratar_informações(tabela.get(comando))
else:
    print(f"O comando {comando} não existe.")
```

O elemento comando recebe o que o usuário digitou, onde em seguida, é feita uma comparação do comando com a tabela, para verificar se o comando que o usuário digitou existe ou não, caso o comando exista, ele será de certa forma, “retirado” da tabela através do método .get(), e em seguida, será enviado para a função que irá realizar o tratamento dessa informação. Caso o comando digitado pelo usuário não conste no conjunto, será retornada uma mensagem informando ao usuário que o comando em questão não existe no sistema.

3 CONCLUSÃO

A realização do presente estudo possibilitou abranger novas formas de desenvolvimento estrutural, como desenvolver e tratar informações de um determinado conjunto de texto para execução de funções. Possibilitou também entender como funciona e como utilizar alguns métodos da linguagem Python, em especial os métodos split() e list().