Cibersegurança Ofensiva

Automação de Segurança da Informação com Python

02 – Estruturas de Dados em Python

Agenda

1.1 Estruturas de Dados em Python

1.2 Criando Listas, Tuplas, Conjuntos, Ranges

1.3 Interagindo com Dicionários

1.4 Criando um jogo com o conteúdo dos módulos 1 e 2

1.5 Funções, Classes, Objetos, Métodos

1.6 Práticas

LEONARDO LA ROSA

Leonardo La Rosa

Bio

LEONARDO LA ROSA

Mais de 25 Anos de Experiência nas áreas de TI e Cibersegurança, com atuação em diversos setores de mercado.

Tecnólogo em Processamento de Dados pela UNIBAN
MBA em Gestão de Tecnologia da Informação pela FIAP

● C|EI ● SANS Foundations ● C|SCU ● N|SF ● C|ND ● C|EH MASTER ● C|SA ● E|CIH ● C|TIA ● CASE JAVA ● Lead Implementer ISO27701

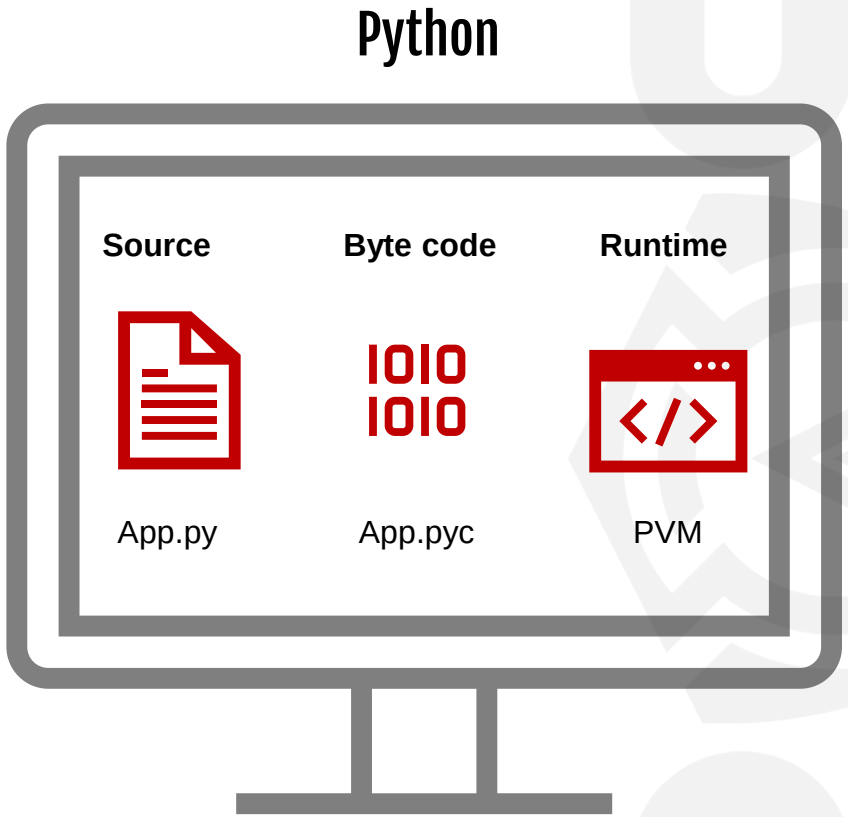
- Cyber Security & Infrastructure Manager
- Docente na Pós Graduação de Cibersegurança
- Instrutor Certificado EC-Council
- Criador de conteúdo e Instrutor de treinamentos personalizados
- Speaker & Digital Influencer

POS | ACADI-TI

Introdução

Diferentes de muitas linguagens de programação, ***Python*** é uma linguagem interpretada, orientada a objetos, de alto nível e com semântica dinâmica.

Por ser uma linguagem simples, ***Python*** reduz a manutenção dos programas, além de suportar módulos e pacotes, que encoraja a programação modularizada e reuso de códigos.



Objetivo do módulo

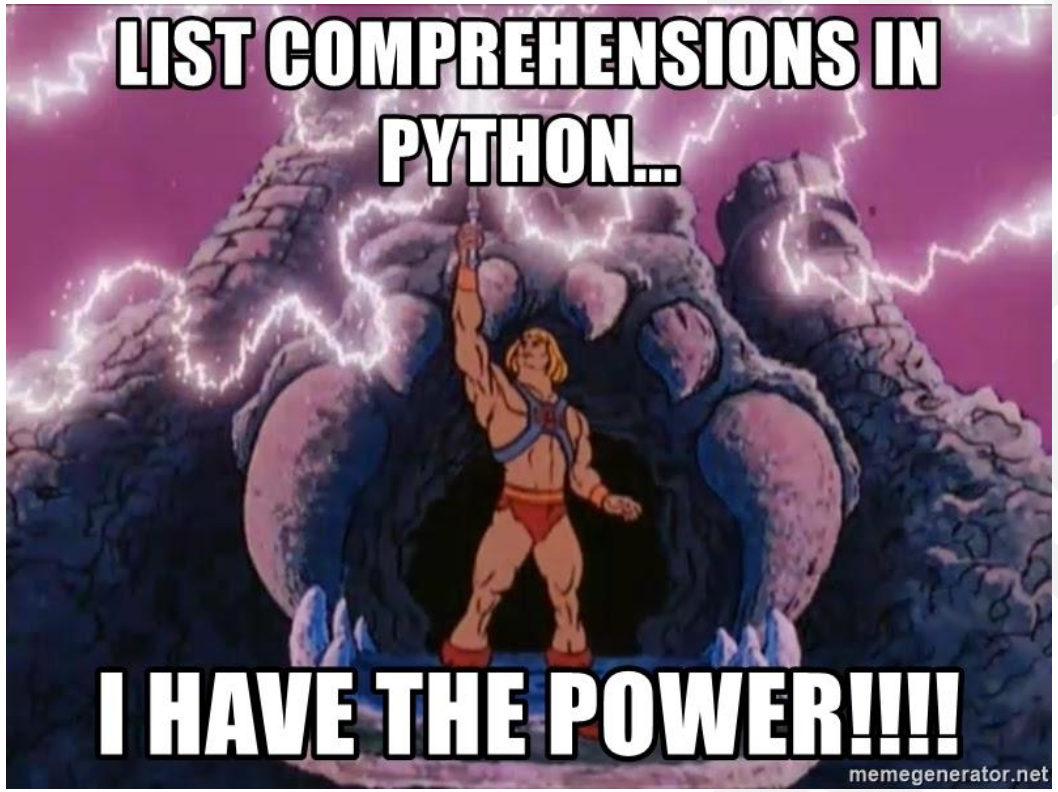
1. Estrutura de Dados em Python
2. Lista
3. Tuplas
4. Range
5. Conjuntos
6. Dicionários
7. Atividades

Estrutura de dados em Python

Estrutura de dados em PYTHON

Neste módulo, abordaremos o tema de estrutura de dados em python e conheceremos tipos de dados que são Sequências.

Sequências são dados que armazenam outros dados, como listas, ranges e tuplas.



Estrutura de dados em PYTHON

Sequências

Existem dois tipos de sequência em Python: **SIMPLES** e **CONTAINER**.

Sequências simples

- Armazenam itens de um só tipo de dado
- Armazenam o valor de cada item em seu próprio espaço de memória

Uma string (ou ***str*** em Python) é um exemplo de sequência simples:

```
curso_pos = ‘Desenvolvimento Python'
```

No exemplo acima, a variável curso_pos armazena em sua memória os valores que são do mesmo tipo (***str***).

Estrutura de dados em PYTHON

Sequências Container

- Podem armazenar itens de diferentes tipos
- Não armazenam o valor de seus itens em sí, mas sim suas referências

Uma lista (ou ***list*** em Python) é um exemplo de sequência container:

```
lista_aluno =['L', 41, 1.73]
```

No exemplo acima estamos armazenando três tipos diferentes: **str**, **int** e **float**. Certo? **ERRADO**! Estamos na verdade armazenando a referência (ou ponteiro) de cada um dos itens.

Estrutura de dados em PYTHON

Sequências Mutáveis e Imutáveis

As sequências também podem ser classificadas entre mutáveis e imutáveis, ou seja, podem ser modificadas ou não após a sua criação

Sequências Mutáveis

- *list, dict*

Observe a lista abaixo contendo 3 letras

```
>>> lista_letras = ['A', 'B', 'C']
>>> print(lista_letras)
['A', 'B', 'C']
```

Podemos modificá-la acrescentando mais 1 letra

```
>>> lista_letras.append('D')
>>> print(lista_letras)
['A', 'B', 'C', 'D']
```

Estrutura de dados em PYTHON

Sequências imutáveis

- *str, tuple, frozenset*

Peraí... Mas eu altero string! Será?

Sempre que você altera uma string, você está na verdade criando outra!

Você não alterou o mesmo objeto! Você criou outra String e atribuiu a referência dela à variável ***nome***

```
>>> nome = "Leonardo"
>>> print(nome)
Leonardo
>>> id(nome)
2774815816112
>>> nome = "La Rosa"
>>> print(nome)
La Rosa
>>> id(nome)
2774815817072
```

Listas

- Uma lista é uma sequência de valores onde cada valor é identificado por um índice iniciado por 0. São similares a strings (coleção de caracteres) exceto pelo fato de que os elementos de uma lista podem ser de qualquer tipo. A sintaxe é simples, listas são delimitadas por colchetes e seus elementos separados por virgula:

Características: Os itens da lista são ordenados, mutáveis e permitem valores duplicados.

```
>>> minha_lista = [1,2,3,4]
>>> print(minha_lista)
[1, 2, 3, 4]
>>> minha_lista.append('cinco')
>>> minha_lista
[1, 2, 3, 4, 'cinco']
>>> minha_lista[1]
2
>>> minha_lista[4]
'cinco'
>>> minha_lista[0]
1
```

Ao tentarmos buscar um valor fora do range da lista, recebemos uma mensagem de erro do interpretador

```
>>> minha_lista[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Listas

- Python permite passar valores negativos, que retornará o valor de uma consulta reversa:

```
>>> minha_lista
[1, 2, 3, 4, 'cinco’]
>>> minha_lista[-1]
'cinco'
>>> minha_lista[-2]
4
>>> minha_lista[-3]
3
```

Também é possível usar a função list() para criar uma lista passando um tipo que pode ser iterável como uma string:

```
>>> lista_nome=list(‘Leonardo’)
>>> lista_nome
['L', 'e', 'o', 'n', 'a', 'r', 'd', 'o']
```

Listas

- Aplicação de uma lista em uma aplicação

```
meses = ['Janeiro', 'Fevereiro', 'Março', 'Abril', 'Maio', 'Junho', 'Julho', 'Agosto', 'Setembro' ,'Outubro',
'Novembro', 'Dezembro']
mes = int(input("Informe o mês do seu nascimento (1-12): "))
if 1 <= mes < 13:
  print('Você nasceu no mês de {}'.format(meses[mes-1]))
```

```
Informe o mês do seu nascimento (1-12): 10
Você nasceu no mês de Outubro
```

Listas

- **Tornando nosso código mais elaborado:**

```
meses = ['Janeiro', 'Fevereiro', 'Março', 'Abril', 'Maio', 'Junho', 'Julho', 'Agosto', 'Setembro' ,'Outubro',
'Novembro', 'Dezembro']
n = int(input("Quantos amigos você tem? "))
amigos = []
for i in range(0,n):
    amigo = input("Informe o nome do amigo {}: ".format(str(i+1)))
    amigos.append(amigo)
for i in range(0,n):
    mes = int(input("Informe o mês de aniversário de {} 1-12: ".format(amigos[i])))
    if 1 <= mes < 13:
        print('{} nasceu no mês de {}'.format(amigos[i],meses[mes-1]))
```

```
Quantos amigos você tem? 2
Informe o nome do amigo 1: Sandy
Informe o nome do amigo 2: Junior
Informe o mês de aniversário de Sandy 1-12: 2
Sandy nasceu no mês de Fevereiro
Informe o mês de aniversário de Junior 1-12: 5
Junior nasceu no mês de Maio
```

Listas

- **Podemos também acessar múltiplos valores em nossa lista, através de fatiamento**

```
meses = ['Janeiro', 'Fevereiro', 'Março', 'Abril', 'Maio', 'Junho', 'Julho', 'Agosto', 'Setembro' ,'Outubro',
'Novembro', 'Dezembro']
print("O primeiro mês do ano é {}".format(meses[:1]))
print("O primeiro trimestre do ano é composto pelos meses: {}".format(meses[:3]))
print("O terceiro trimestre do ano é composto pelos meses: {}".format(meses[6:9]))
print("O primeiro semestre é composto pelos meses: {}".format(meses[0:6]))
print("O segundo semestre é composto pelos meses: {}".format(meses[6:]))
```

```
O primeiro mês do ano é ['Janeiro']
O primeiro trimestre do ano é composto pelos meses: ['Janeiro', 'Fevereiro', 'Março']
O terceiro trimestre do ano é composto pelos meses: ['Julho', 'Agosto', 'Setembro']
O primeiro semestre é composto pelos meses: ['Janeiro', 'Fevereiro', 'Março', 'Abril', 'Maio', 'Junho']
O segundo semestre é composto pelos meses: ['Julho', 'Agosto', 'Setembro', 'Outubro', 'Novembro', 'Dezembro']
```

Listas

- **As listas também possuem funcionalidades prontas, e podemos manipulá-las através de funções embutidas**

```
>>> lista_numero=[]
>>> lista_numero.append(1)
>>> lista_numero.append('dois')
>>> lista_numero
[1, 'dois']
```

A função append() só consegue inserir um elemento por vez.

- Se quisermos inserir mais elementos, podemos somar, multiplicar ou utilizar a função extend() :

```
>>> lista_numero=[1,2,"tres",4]
>>> lista_numero.extend(['cinco',6])
>>> lista_numero += ['sete',8]
>>> lista_numero + [9]
[1, 2, 'tres', 4, 'cinco', 6, 'sete', 8, 9]
>>> lista_numero *2
[1, 2, 'tres', 4, 'cinco', 6, 'sete', 8, 1, 2, 'tres', 4, 'cinco', 6, 'sete', 8]
```

Tuplas

- Uma tupla é uma sequência que não pode ser alterada depois de criada. Uma tupla é definida de forma parecida com uma lista com a diferença do delimitador. Enquanto listas utilizam colchetes como delimitadores, as tuplas usam parênteses:

Características: Os itens da tupla são ordenados e imutáveis

```
>>> minha_tupla = ('um', 'dois', 'tres')
>>> type(minha_tupla)
<class 'tuple'>
```

```
>>> minha_tupla2 = (1, 2, 3)
>>> type(minha_tupla2)
<class 'tuple'>
```

Assim como as listas, também podemos usar uma função para criar uma tupla passando um tipo que pode ser iterável como uma string ou uma lista. Essa função é a tuple() :

```
>>> nome = "Leonardo"
>>> tuple(nome)
('L', 'e', 'o', 'n', 'a', 'r', 'd', 'o')
```

```
>>> numeros = [1, 2, 3, 4]
>>> tuple(numeros)
(1, 2, 3, 4)
```

Tuplas

- Como são imutáveis, uma vez criadas não podemos adicionar nem remover elementos de uma tupla

```
>>> minha_tupla=(1, 2, 3, 4)
>>> minha_tupla.append(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```

Observe que o objeto tupla (tuple) não possui o atributo append().

```
>>> minha_tupla=(1, 2, 3, 4)
>>> minha_tupla[0] = 'um'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Não é possível atribuir valores aos itens individuais de uma tupla.

Tuplas

- Não é possível atribuir valores aos itens individuais de uma tupla. no entanto, é possível criar tuplas que contenham objetos mutáveis, como listas.

```
>>> minha_lista = ["quatro", "cinco"]
>>> minha_tupla = ('um', 'dois', 'tres', minha_lista)
>>> minha_tupla
('um', 'dois', 'tres', ['quatro', 'cinco'])
```

As tuplas são imutáveis e geralmente contêm uma sequência heterogênea de elementos. Já as listas são mutáveis, e seus elementos geralmente são homogêneos, sendo acessados pela iteração da lista, embora não seja uma regra.

Range

- O range é um tipo de sequência de números, sendo comumente usado para looping de um número específico de vezes em um comando for já que representam um intervalo.

Características: Os itens do Set são imutáveis

```
range(inicio, fim)
```

```
>>>sequencia = range(1,   3)
>>>print(sequencia)
range(1,  3)
```

O range não imprime os elementos da sequência, ele apenas armazena seu início e seu final. Para imprimir seus elementos precisamos de um laço for :

```
>>> for sequencia in range(1,3):
...    print(sequencia)
...
1
2
```

O número finalizador, o fim, não é incluído na sequência. Vejamos um exemplo:

Range

- O Range também possibilita a inclusão de um outro parâmetro, que é a variação entre um número e seu sucessor

```
>>> for lista_2 in range(0,20,2):
...  print(lista_2)
...
0
2
4
6
8
10
12
14
16
18
```

```
>>> for lista_5 in range(0,50,5):
...  print(lista_5)
...
0
5
10
15
20
25
30
35
40
45
```

Range

- Vamos criar uma aplicação que realiza a tabuada do 1 ao 10 e exibe os valores na tela

```
tabuada = int(input("Qual a tabuada você deseja exibir?: "))
for i in range(1,11):
    print("{} x {} = {}".format(i,tabuada,(i*tabuada)))
    i+1
```

```
Qual a tabuada você deseja exibir?: 5
1 x 5 = 5
2 x 5 = 10
3 x 5 = 15
4 x 5 = 20
5 x 5 = 25
6 x 5 = 30
7 x 5 = 35
8 x 5 = 40
9 x 5 = 45
10 x 5 = 50
```

SET (conjuntos)

- Um conjunto é uma coleção que é ao mesmo tempo desordenado e desindexado. Diferente de uma lista ou uma tupla, Os conjuntos são escritos entre chaves "{}".

Características: Os itens do conjunto não são ordenados, são imutáveis e não permitem valores duplicados.

```
>>> frutas = {'banana', 'maçã', 'uva'}
>>> frutas
{'banana', 'uva', 'maçã'}
>>> type(frutas)
<class 'set'>
```

Você não pode alterar os itens de um conjunto, mas você pode adicionar ou remover novos itens.

```
>>> frutas.add('abacaxi')
>>> frutas
{'banana', 'uva', 'abacaxi', 'maçã'}
```

```
>>> frutas.remove('abacaxi')
>>> frutas
{'banana', 'uva', 'maçã'}
```

SET (conjuntos)

- Para criar um conjunto vazio você tem que usar set() , não {}

```
>>> dados = {}
>>> type(dados)
<class 'dict'>
```

Criação de um dicionário vazio

```
>>> dados = set()
>>> type(dados)
<class 'set'>
```

Criação de um conjunto vazio

SET (conjuntos)

- Usos de SET incluem testes de associação e eliminação de entradas duplicadas. Os objetos de conjunto também suportam operações matemáticas como união, interseção, diferença e diferença simétrica. Podemos transformar um texto em um conjunto com a frunção set() e testar os operações:

```
>>> a = set('cavalo')
>>> b = set('cachorro')
>>> a
{'c', 'v', 'a', 'o', 'l'}
>>> b
{'c', 'a', 'o', 'h', 'r'}
>>> a - b
{'v', 'l'}
>>> a | b
{'c', 'v', 'a', 'o', 'h', 'r', 'l'}
>>> a & b
{'o', 'c', 'a'}
>>> a ^ b
{'h', 'r', 'v', 'l'}
```

```
>>> a = set('cavalo')
>>> b = set('cachorro')
>>> a
{'c', 'v', 'a', 'o', 'l'}
>>> b
{'c', 'a', 'o', 'h', 'r'}
>>> a.difference(b)
{'v', 'l'}
>>> a.union(b)
{'c', 'v', 'a', 'o', 'h', 'r', 'l'}
>>> a.intersection(b)
{'o', 'c', 'a'}
>>> a.symmetric_difference(b)
{'h', 'r', 'v', 'l'}
```

diferença

união

Intersecção

Diferença Simétrica

Dicionários

- Dicionário é outra estrutura de dados em Python e seus elementos, sendo estruturadas de forma não ordenada assim como os conjuntos.

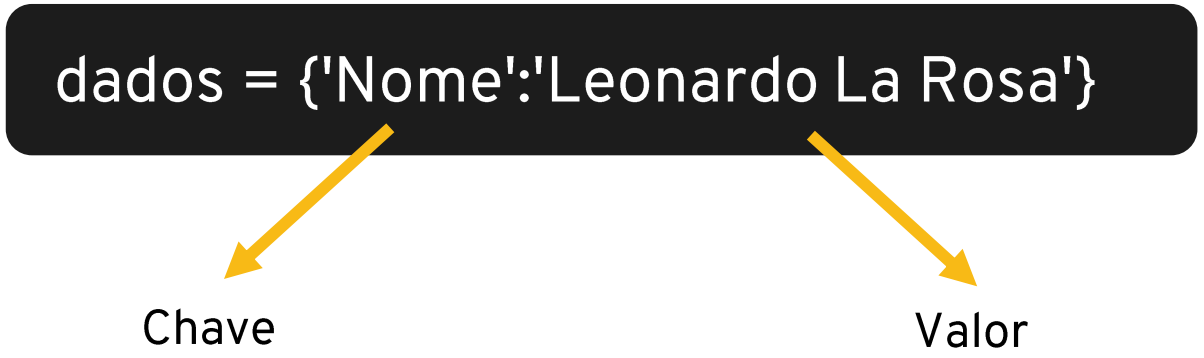
Características: Os itens do dicionário são ordenados, mutáveis e não permitem valores duplicados.

Os dicionários são estruturas poderosas e muito utilizadas, já que podemos acessar seus elementos através de chaves e não por sua posição. Em outras linguagens, este tipo é conhecido como "matrizes associativas".

```
>>> dados = {'Nome':'Leonardo La Rosa', 'Idade':41 , 'Cidade':'São Paulo'}
>>> dados
{'Nome': 'Leonardo La Rosa', 'Idade': 41, 'Cidade':'São Paulo'}
>>> type(dados)
<class 'dict'>
```

Dicionários

- Os dicionários são delimitados por chaves ({}) e suas chaves (‘chave1', ‘chave2' e ‘chave3') por aspas. Já os valores podem ser de qualquer tipo. No exemplo acima, temos duas strings e um int .



- Diferente das Listas, Tuplas e Conjuntos, os elementos em um dicionário são acessados pelas suas chaves ou valores

```
>>> dados = {'Nome':'Leonardo La Rosa'}
>>> dados[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 0
```

```
>>> dados = {'Nome':'Leonardo La Rosa'}
>>> dados['Nome']
'Leonardo La Rosa'
```

Dicionários

- O dicionário possui um método chamado keys() que devolve o conjunto de suas chaves:

```
>>> dados = {'Nome':'Leonardo La Rosa', 'Idade':41, 'Cidade':'São Paulo'}
>>> dados.keys()
dict_keys(['Nome', 'Idade', 'Cidade'])
```

- Assim como um método chamado values() que retorna seus valores:

```
>>> dados = {'Nome':'Leonardo La Rosa', 'Idade':41, 'Cidade':'São Paulo'}
>>> dados.values()
dict_values(['Leonardo La Rosa', 41, 'São Paulo'])
```

Dicionários

- Somente tipos de dados imutáveis podem ser usados como chaves, ou seja, não podem ser usadas listas, porém, as tuplas são permitidas

```
>>> lista = ['a','b','c']
>>> type(lista)
<class 'list'>
>>> dic = {lista:2}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

```
>>> tupla = ('a','b','c')
>>> type(tupla)
<class 'tuple'>
>>> dic = {tupla:2}
>>> type(dic)
<class 'dict'>
```

- Outra forma de criarmos dicionário é usando a funão dict()

```
>>> dados = dict(Nome='Leonardo', Idade=41,
Cidade='São Paulo')
>>> dados
{'Nome': 'Leonardo', 'Idade': 41, 'Cidade': 'São Paulo'}
```

Atividade

Vamos criar o jogo da forca, usando o conhecimento que adquirimos até aqui, com estrutura de decisões, listas, set e loops.


```
def jogar():
    letras_usadas = set()  #Criamos um Conjunto pois não queremos que as letras repetidas sejam exibidas
    palavra_secreta = input("Digite a palavra para ser descoberta: ")
    print("\n" * 100) #limpamos a tela para que o jogador não leia a palavra
    print('*********************************')
    print('***Bem vindo ao jogo da Forca!***')
    print('*********************************')
    letras_acertadas = []  #Criamos uma lista, que será alimentada com a quantidade de letras da palavra secreta
    for x in range(1,len(palavra_secreta)+1): #usando o loop for, identificaremos com o range o tamanho da minha palavra secreta
        letras_acertadas.append("_") #em seguida, acrescentaremos um "_" para que o jogador saiba quantas letras a palavra possui
        x+1 #a cada letra analisada, o contador é incrementado
    enforcou = False #adicionamos o valor boleano para enforcou e acertou, para determinar quando o jogo termina
    acertou = False
    erros = 0 #definimos o valor de erro para 0
    print(' '.join(letras_acertadas)) #removemos as aspas e ","... DE: ['_','_'] PARA: _ _
```

Atividade

```
    while not enforcou and not acertou: #enquanto não enforcou e não acertou, o jogo continua
        chute = input("Digite uma letra: ") # o jogador pode digitar mais uma letra
        if chute.upper() in palavra_secreta.upper(): #para letras maiúscula e minúsculas serão aceitas, usamos o método .upper()
            posicao = 0
            for letra in palavra_secreta.upper(): #cada vez que a letra for encontrada, o "_" na posição é substituído.
                if chute.upper() == letra.upper():
                    letras_acertadas[posicao] = letra
                posicao = posicao + 1
        else:
            erros += 1 #se a letra estiver incorreta, o contador de erro é incrementado
        enforcou = erros == 6 # após 6 erros, o jogo termina
        acertou = '_' not in letras_acertadas
        print(' '.join(letras_acertadas))
        letras_usadas.add(chute.upper()) #as letras usadas são adicionadas e exibidas
        print("Letras Usadas: {}".format(letras_usadas))
    if acertou:
        print('Você ganhou!!')
    else:
        print('Você perdeu!!')
    print('Fim do jogo')
jogar()
```

Funções

- Como vimos em nosso exercício, o jgo é iniciado quando chamamos a função jogar(). Uma função é um bloco de código que só funciona quando é chamado. Você pode passar dados, conhecidos como parâmetros, para uma função e ela pode retornar dados como resultado.

```
def minha_funcao():
   print("Olá Mundo!")
minha_funcao()
```

```
Olá Mundo!
```

```
def minha_funcao(fname):
  print(fname + " é um Acadiano")
minha_funcao("Leonardo")
minha_funcao("Leandro")
```

```
Leonardo é um Acadiano
Leandro é um Acadiano
```

Função Lambda

- Uma função lambda é uma pequena função anônima. Uma função lambda pode ter qualquer número de argumentos, mas só pode ter uma expressão.

```
flambda = lambda a : a + 100
print(flambda(5))
```

```
105
```

```
flambda = lambda a, b, c : a * b + c
print(flambda(10, 3, 2))
```

```
32
```

Classes e Objetos

- Quase tudo em Python é um objeto, com suas propriedades e métodos. Uma Classe é como um construtor de objetos, ou um "projeto" para criar objetos.

Podemos realizar uma analogia entre o projeto de uma casa (a planta da casa) e a casa em si. O projeto é a **classe** e a casa, construída a partir desta planta, é o **objeto**.

Classes e Objetos

- Exemplo de criação de uma Classe e um Objeto.

Classe

```
class MinhaClass:
  nome = "Leonardo"
  idade = 41
```

Criamos uma classe com duas propriedades: nome e idade

Objeto

```
pessoa = MinhaClass()
print(pessoa.nome)
print(pessoa.idade)
```

Em seguida, utilizamos a classe MinhaClass() para criar nosso objeto pessoa

```
Leonardo
41
```

A função __init__()

- O exemplo da página anterior são classes e objetos em sua forma mais simples, e não são realmente úteis em aplicações da vida real. Todas as Classes têm uma função chamada __init__(), que é sempre executada quando a Class está sendo iniciada.

Classe

```
class Pessoa:
  def __init__(self, nome, idade):
    self.nome = nome
    self.idade = idade
```

Objeto

```
p1 = Pessoa("Leonardo", 41)
print(p1.nome)
print(p1.idade)
```

```
Leonardo
41
```

Métodos e Objetos

- Objetos também podem conter métodos. Métodos em objetos são funções que pertencem ao objeto.


```
class Pessoa:
  def __init__(self, nome, idade):
    self.nome = nome
    self.idade = idade
  def minhafuncao(self):
    print("Olá, meu nome é {} e tenho {} anos.".format(self.nome, self.idade))
```

```
p1 = Pessoa("Leonardo", 41)
p1.minhafuncao()
```

```
Olá, meu nome é Leonardo e tenho 41 anos.
```

Métodos e Objetos

- Você também pode alterar ou deletar alguma propriedade de um objeto ou mesmo o próprio objeto.

```
p1.idade = 50
p1.minhafuncao()
```

```
Olá, meu nome é Leonardo e tenho 50 anos.
```

```
del p1.idade #deleta a propriedade idade
p1.minhafuncao()
```

```
AttributeError: 'Pessoa' object has no attribute 'idade'
```

```
del p1 #deleta  objeto
p1.minhafuncao()
```

```
NameError: name 'p1' is not defined
```

Módulos

- Considere um módulo como o mesmo que uma biblioteca de códigos. Um arquivo contendo um conjunto de funções que você deseja incluir em seu aplicativo.


```
def greeting(nome):
  print("Olá, " + nome)
```

Salvar como modulo.py

```
>>> import modulo
>>> modulo.greeting("Leonardo")
Olá, Leonardo
```

```
Pessoa = {
  "nome": "Leonardo",
  "idade": 41,
  "pais": "Brasil"
}
```

Salvar como modulo.py

```
>>> import modulo
>>> x = modulo.pessoa["idade"]
>>> y = modulo.pessoa["nome"]
>>> z = modulo.pessoa["pais"]
>>> print("Olá, meu nome é {}, tenho {} anos e moro no {}.".format(y,x,z))
Olá, meu nome é Leonardo, tenho 41 anos e moro no Brasil.
```

Módulos

- Podemos também usar um Alias para importar nosso módulo.

```
Pessoa = {
  "nome": "Leonardo",
  "idade": 41,
  "pais": "Brasil"
}
```

Salvar como modulo.py

```
>>> import modulo as abc
>>> x = abc.pessoa["idade"]
>>> y = abc.pessoa["nome"]
>>> z = abc.pessoa["pais"]
>>> print("Olá, meu nome é {}, tenho {} anos e moro no {}.".format(y,x,z))
Olá, meu nome é Leonardo, tenho 41 anos e moro no Brasil.
```

Módulos – Função dir()

- Podemos utilizar a função built-in dir() para listar todos os nomes de funções ou variáveis em um módulo:

```
Pessoa = {
  "nome": "Leonardo",
  "idade": 41,
  "pais": "Brasil"
}
```

Salvar como modulo.py

```
>>> print(dir(modulo))
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__', 'pessoa']
```

- A partir daí, podemos importar a parte do módulo que nos interessa:

```
Pessoa = {
  "nome": "Leonardo",
  "idade": 41,
  "pais": "Brasil"
}
```

```
>>> from modulo import pessoa
>>> print(pessoa["nome"])
Leonardo
```

Praticar

Para praticar, durante a semana, recomendo os exercícios que são propostas no link:

https://www.w3schools.com/python/exercise.asp.

No total são 95 exercícios abordando todos os tópicos do módulo 1 e 2

Praticar

Para praticar, durante a semana, recomendo os exercícios que são propostas no link:

https://www.w3schools.com/python/exercise.asp.

No total são 95 exercícios abordando todos os tópicos do módulo 1 e 2

Referências Bibliográficas

- Caelum. Escola de Tecnologia, accessed: August 2021, https://www.caelum.com.br/apostila-python-orientacao-a-objetos/

- Reitz, Kenneth and Schlusser, Tanya, O Guia do moclileiro Python: 2017, https://www.amazon.com.br/Guia-Mochileiro-Python-Melhores-Desenvolvimento/dp/8575225413/ref=asc_df_8575225413/?tag=googleshopp00-20&linkCode=df0&hvadid=379765802639&hvpos=&hvnetw=g&hvrand=15526091200442465043&hvpone=&hvptwo=&hvqmt=&hvdev=c&hvdvcmdl=&hvlocint=&hvlocphy=1001773&hvtargid=pla-811121403521&psc=1.

- Allen B. Downey, Pense Python, 2016, https://novatec.com.br/livros/pense-em-python/

- Brandon Rhodes, John Goerzen, 2015, https://novatec.com.br/livros/programacao-redes-com-python/#:~:text=Ver%20mais%20%E2%96%BC-,Programa%C3%A7%C3%A3o%20de%20redes%20com%20Python%20aborda%20todos%20os%20t%C3%B3picos%20cl%C3%A1ssicos,as%20atualiza%C3%A7%C3%B5es%20de%20Python%203.

- Python Tutorial, Accessed: September, 2021, Python Tutorial (w3schools.com)

POS ACADI-TI

OBRIGADO

Desenvolvimento Seguro