MATERIAL COMPLEMENTAR

PYTHON PARA PENTESTERS

Escrito por Mario Cavalheiro | COO
Desec Security

INTRODUÇÃO

Atuar como Pentester é para quem **gosta de desafios**, quem gosta de levar o conhecimento ao limite nos mais adversos cenários encontrados e então aprender ainda mais para encontrar uma solução para o problema. Um poderoso kit de ferramentas é sempre um forte aliado durante estes processos, e melhor ainda se for ferramentas de sua própria criação!

A utilização de ferramentas em Pentest é muito comum seja para automatizar um processo ou realizar algum teste específico. Existem inúmeras ferramentas públicas disponíveis, porém em alguns momentos durante o Pentest o cenário não permite a utilização por faltar dependências obrigatórias, o que leva o Pentester precisar desenvolver algo rápido. Neste ponto tempos o Python como forte aliado.

O Python é uma linguagem multi-plataforma de alto nível, interpretada, criada por Guido Van Rossaum com o objetivo de ser facilmente legível, criar maior produtividade devido a fácil implementação pela imensa variedade de recursos disponíveis para realizar quase qualquer tarefa necessária de um modo rápido e facilmente legível.

OBJETIVO

Este tutorial tem como objetivo demonstrar os principais recursos para a lógica de programação utilizando a sintaxe do Python 3. O foco é ensinar a escrita, leitura e interpretação do que os códigos fazem de um jeito simples e prático. O código proposto trabalhará a estruturação e lógica, apresentando assim a base necessária para que você saiba criar suas ferramentas em Python, ou mesmo editar códigos em Python.

Os exemplos serão sempre abordados de forma prática, com um nível de dificuldade básico. Como o intuito não é a eficiência total do código em execução e sim ensinar, não será visto nada avançado e todos os processos serão criados manualmente do modo mais simples possível.

O *script* proposto trabalhará a estruturação e lógica, apresentando assim os principais recursos necessários para que você consiga criar suas ferramentas em Python. Ao final teremos um modulo adicional que implementaremos funcionalidades ao script, dando assim atribuições práticas a ele.

PROPOSTA

Como proposta de código a ser estudado, vamos ver do início ao fim a estruturação de um *Port Scanning*. Lembrando que o código será simples, porém com ele será aprendido a base do Python.

Buscaremos expor as funcionalidades da forma mais prática possível, porém terão alguns conceitos básicos que precisam de um pouco mais de teoria para entender o que está havendo. Sabendo disso, buscaremos explicar da forma mais direta possível o conceito necessário.

ESTRUTURA DO CÓDIGO

Identação

Assim como em qualquer linguagem o Python possui regras a serem seguidas, a fim do código ser corretamente interpretado e executado corretamente. Uma das principais características do Python é a identação.

Muitas linguagens utilizam caracteres especiais para as delimitações de final de linha, ou escopo de um bloco de código. O Python utiliza espaços, e estes espaços são a Identação!

Para realizar a identação é comum utilizar quatro espaços ou uma tabulação. Os códigos em Python são interpretados de cima para baixo, da esquerda para a direita. Códigos ao lado extremo da esquerda estão na primeira linha hierárquica, e são os primeiros a serem executados, já os com identação estão nas hierarquias posteriores.

A delimitação inicia-se na margem, caso seja feito uma tabulação as próximas hierarquias devem seguir 2, 3, 4 tabulações e assim sucessivamente. O mesmo ocorre com espaços, se o primeiro for realizado utilizando quatro espaços os seguintes deverão ter 8, 12, 16... e assim por diante, devendo manter-se constante.

Identação **Errada**:

```
def exemplo(valor):
try:
valor.x = input('Apenas um exemplo...')
x = valor.x
return x
except:
print 'Fim.'
valor.exemplo()
```

Identação **Correta**:

```
def exemplo(valor):
    try:
        valor.x = input('Apenas um exemplo...')
        x = valor.x
        return x
    except:
        print 'Fim.'
        valor.exemplo()
```

Visivelmente os dois códigos são iguais e corretos (fictícios, apenas para exemplo). Porém o que está com a identação errada não é interpretado e irá gerar erro.

Será utilizado a identação de 4 espaços nos exemplos desse tutorial.

Blocos de Instrução

Um bloco de instrução é composto por diversas instruções no mesmo nível de identação. Ou seja, códigos que estão na mesma identação podem ser chamados de um Bloco de Instrução.

No exemplo anterior temos o caso do Try, onde os códigos na função possuem a mesma hierarquia:

```
def exemplo(valor):
    try:
        valor.x = input('Apenas um exemplo...')
        x = valor.x
        return x
    except:
        print 'Fim.'
        valor.exemplo()
```

INTERPRETAÇÃO DA LINGUAGEM

O Python possui uma **linguagem interpretada**, e possui seu próprio interpretador. A função do interpretador é ler o código fonte e realizar a conversão "em tempo real" para uma linguagem de baixo nível (os 0 e 1...) para que a máquina possa entender e processar corretamente os dados.

O fluxo do interpretador é conforme o código, onde ele lê linha a linha, verifica se a sintaxe está correta e então converte para a linguagem de máquina para executar as instruções. Este é uma das razões da qual muitas vezes executamos um script em python e ele apenas alerta o erro no decorrer do processo.

Existem dois modos para executarmos scripts em python: **arquivos .py**, ou utilizando o **modo interativo**.

Arquivos .py

A extensão .py simboliza que o arquivo se trata de um script em Python. Dessa forma podemos executar o script passando a referência na primeira linha dele de onde o interpretador está localizado:

```
root@desecsecurity:~/python# ls
script.py
```

```
#!/usr/bin/python3
print("Learning Python - Desec Security")
```

Aqui podemos utilizar o comando chmod +777 para dar permissões de execução no arquivo diretamente pelo comando "./":

```
root@desecsecurity:~/python# chmod +777 script.py
root@desecsecurity:~/python# ls
script.py
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security
```

Podemos ainda apenas invocar o interpretador "python3" para executar o código. Dessa forma não é preciso dar permissão de execução, e nem informar onde está localizado o interpretador na primeir alinha do código:

```
root@desecsecurity:~/python# ls
script.py
root@desecsecurity:~/python# cat script.py
print("Learning Python - Desec Security")
root@desecsecurity:~/python# python3 script.py
Learning Python - Desec Security
```

Python IDLE

Ambiente Integrado de Desenvolvimento e Aprendizagem (Integrated Development and Learning Environment) é um recurso que o interpretador Python disponibiliza além de ler arquivos .py.

Podemos invoca-lo e executar comandos, normalmente é utilizado para pequenas ações:

```
root@desecsecurity:~/python# python3
Python 3.7.4 (default, Jul 11 2019, 10:43:21)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Learning Python - Desec Security")
Learning Python - Desec Security
>>> 27*6
162
>>> port = 80
>>> print(port)
80
```

No exemplo acima vemos que é possível realizar várias operações. Os comandos são enviados logo após os sinais “>>>” e interpretados em seguida, ou também podemos armazenar os valores em variáveis para utilização posterior (Veremos sobre variáveis nos tópicos seguintes).

Para não ser criado vários arquivos durante as explicações, muitos dos conceitos na prática serão vistos primeiramente utilizando o IDLE, e posteriormente aplicados em scripts com a extensão “.py”. Dessa forma podemos explorar diversos aspectos das funcionalidades do que será visto de um modo rápido, podendo ser replicado facilmente.

ESTRUTURAÇÃO DO CÓDIGO

INPUT / OUTPUT

Saída de Dados - Output

Tudo que é exibido na tela é considerado um Output seja uma simples mensagem, ou o resultado de uma determinada ação que o código realizou.

Para criarmos um **Output** de mensagem utilizaremos a função “print()”.

Sua estrutura é bem simples, para uma saída de texto basta informar a mensagem desejada entre duas aspas:

print(“Mensagem a ser exibida”)

./script.py:

```
#!/usr/bin/python3
print("Learning Python - Desec Security")
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security
```

Quando utilizamos uma saída de texto na tela o Python irá entender que não são comandos e sim apenas uma mensagem. Com isso, tudo que estiver entre aspas será definido apenas como texto, e não como algo a ser executado (algum comando ou ação).

Porém podemos contar com alguns caracteres especiais que trabalham unicamente para auxiliar a melhor exibição do texto, sendo eles: n, t, r.

Para utilizarmos cada um é necessário que seja inserido uma contrabarra (backslash “\”) seguido da letra. Assim, o interpretador irá verificar se na mensagem de texto contém um backslash, se conter ele irá executar a ação representante da letra correspondente.

\n	Realiza a quebra de linhas no texto (**New line**)
\t	Realiza a insersão de uma tabulação (**Tab**)

Por exemplo, ao utilizarmos na prática em nosso Output alguns desses recursos temos a seguinte saída:

```
#!/usr/bin/python3
print("Learning\tPython \n- \nDesec Security")
```

```
root@desecsecurity:~/python# ./script.py
Learning        Python
-
Desec Security
```

Uma outra característica bastante interessante nas linguagens de programação é a possibilidade de adicionarmos **comentários!**

Comentários

Os comentários servem para tornar o código mais organizado para o programador. Podemos (e devemos pelas boas práticas) utilizar os comentários para oferecer instruções sobre o que está ocorrendo em determinada parte do código.

A sintaxe para criar um comentário no Python 3 é o hashteg (#) como primeiro caractere da linha:

 # Aqui podemos inserir os comentários

```
#!/usr/bin/python3
#Adicionando comentarios
#Abaixo temos a funcao print exibindo uma mensagem.
print("Learning Python - Desec Security")

root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security
root@desecsecurity:~/python#
```

Nenhuma das palavras ou caracteres inseridos após o *hashtag* serão interpretadas pelo Python, não gerando saída na tela como visto no exemplo acima. Comentários são úteis e direcionados apenas para quem está interagindo diretamente com o código.

Por mais que o Python seja bem intuitivo com sua sintaxe limpa, a prática de tornar público os códigos são comuns. Realizar bons comentários com instruções sobre funcionalidades específicas de sua ferramenta poderá auxiliar muito no entendimento de quem a utilizar posteriormente. Os comentários auxiliam também para manter o código, quando compartilhado através do GitHub para contribuições por exemplo.

O objetivo do script não é apenas exibir informações na tela e sim realizar ações através da verificação de dados. Para isso é preciso passar valores a ele através dos chamados **Inputs** (entradas).

Entrada de Dados – Input

Se existe a saída deve haver a entrada! E é isso mesmo, como o próprio termo já diz, a "Saída de dados" são informações exibidas na tela seja através de algum recurso definido como mensagens intuitivas, ou resultado de ações realizadas pelo programa. Já a entrada de dados é o recurso que permite ao programa receber valores a ser trabalhado.

Neste caso utilizaremos entradas através da linha de comando, onde será requisitado que seja inserido através do teclado informações durante a execução do script. Porém, todos os dados que o script coleta, seja durante requisição ou no fluxo de sua execução, ele precisa armazena-los em algum local, e para isso existem as **Variáveis**!

Variáveis

Uma variável é um espaço alocado na memória que servirá para realizar o armazenamento de um valor específico a ser utilizado durante a execução do código. Assim como no português temos letras e números onde cada um é utilizado para um propósito, na programação também existe uma classificação entre cada tipo de valor.

Tipos de Variáveis

Os principais tipos de variáveis que serão utilizadas nas ferramentas ofensivas são:

TIPO	VALORES ARMAZENADOS
int (integer)	Números inteiros.
float (floating)	Números reais.
bool (boolean)	Verdadeiro ou Falso (True / False)
str (string)	Strings (Textos). Armazena todo e qualquer tipo de valor em formato de texto.

A sintaxe de declaração de variável em Python tem uma estrutura de três elementos básicos:

nome_da_variavel = valor_atribuido

Nome_da_variavel: O nome da variável normalmente é escolhido de acordo com o valor que será atribuído. Pelas boas práticas e melhor leitura do código, sugere-se isso.
Sinal de Igual: Diferente do conceito tradicional onde o sinal de igual representa "equivalência", na programação o sinal refere-se à **Atribuição**.

> *Quando existe um sinal de "=" significa que o valor seguinte está sendo **atribuído** a uma variável que está a esquerda, por exemplo quando informamos "**porta = 22**". Informamos que o valor "22" está sendo **atribuído** à variável usuário.*

valor_atribuido: Valor inserido através de entrada do usuário ou coleta do próprio código durante a execução.

Nomes Proibidos

Algumas regras devem ser seguidas para a criação da variável tais como as restrições de nome, onde ele **NÃO PODE**:

- Conter caracteres latinos (ç, ~, acentuação, etc..)
- Conter caracteres especiais (!,#$%*{] …..)
- Iniciar com um número (7variavel..)
- Conter espaçamento (variável usuário)

> *OBS: O único caractere especial que poderá ser utilizado no início de um nome de variável é o "underline" (_), onde neste caso informa que a variável, método, função ou classe é **privada**. Porém este conceito será visto em tópicos adiante.*

Também há as **Palavras Reservadas** da linguagem, que não podem ser utilizadas como nome de variáveis:

> **and**, **as**, **assert**, **break**, **class**, **continue**, **def**, **del**, **elif**, **else**, **except**, **exec**, **False**, **finally**, **for**, **from**, **global**, **if**, **import**, **in**, **is**, **lambda**, **None**, **nonlocal**, **not**, **or**, **pass**, **raise**, **return**, **True**, **try**, **while**, **with**, **yield**.

As palavras reservadas são de uso interno da linguagem, onde cada uma possui funcionalidades pré-definidas para o funcionamento do interpretador. A lista pode sofrer modificações com o tempo e novas implementações na linguagem.

Regras para os Valores atribuídos

Ainda falando sobre a declaração, os **valores atribuídos** às variáveis devem seguir uma regra básica, sendo:

Valores do tipo texto devem ser declarados entre apóstrofos (ou aspas):

```
usuario = 'root'
usuario = "root"
```

Valores numéricos devem ser declarados sem apóstrofos (do contrário, serão considerados texto):

```
porta = 22
```

Variáveis booleanas envolvem um estado, ou é Verdadeiro ou Falso (**True** / **False**):

```
conectado = True
```

> *Veremos nos próximos tópicos conforme o script é incrementado, como podemos utilizar as variáveis booleanas a nosso favor para as tomadas de decisões das ferramentas.*

Observando os tipos de variáveis com o interpretador

A seguir, vamos implementar algumas variáveis no script e ver na prática como é feito esta tipagem:

```
#!/usr/bin/python3
#Abaixo temos a funcao print() exibindo uma mensagem.
print("Learning Python - Desec Security")

ip = '192.168.1.107'
porta = 21
segundos_conectado = 27.18
conectado = False

print("IP:",type(ip))
print("Porta:",type(porta))
print("Tempo em Segundos:",type(segundos_conectado))
print("Conectado:",type(conectado))
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security
IP: <class 'str'>
Porta: <class 'int'>
Tempo em Segundos: <class 'float'>
Conectado: <class 'bool'>
```

Vemos agora uma nova função: ***type*()**.

Função **type()**:

Realiza a verificação do tipo de um valor ou de uma variável, bastanto informa-lo/a como argumento:

type(ip) → Verifica o tipo de dado que a **variável** "ip" contém.
type(21) → Verifica qual é o tipo de classe do **dado** "21".

Em exemplo, a função **type()** foi inserida dentro da função **print()** para uma exibição mais amigável do output. Neste caso não queremos saber do valor que contém cada variável, e sim o seu tipo!

Agora surge um ponto interessante: Como foi que o Python realizou a classificação dos valores inseridos e definiu tipos a eles?

O Python adota um conceito existente na programação chamado de "Duck Typing" (tipagem do pato, literalmente), vinda da expressão por James Whitcomb Riley:

> **"Se ele parece com um pato, nada como um pato e grasna como um pato, então provavelmente é um pato".**

Sabendo desse conceito é possível entender que durante o armazenamento de algum valor em uma variável, o interpretador verifica as "características" do valor que foi atribuído e assim então identifica a classe que melhor se enquadra a ele.

Entrada de Dados na prática

Podemos fazer com que o script requisite valores para o usuário ao invés de passarmos valores pré-definidos. Para isso contamos com o recurso da função Input(). A função Input realiza uma pausa na execução do script e aguarda até que dados sejam digitados e inseridos com Enter.

Estrutura:

nome_da_variavel = input("Mensagem Intuitiva")

```
1 #!/usr/bin/python3
2 #Abaixo temos a funcao print() exibindo uma mensagem.
3 print("Learning Python - Desec Security\n")
4
5 porta = input("Informe uma porta: ")
6 print("\nPorta: ",porta)
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security

Informe uma porta: 21

Porta:  21
```

Perfeito, dessa forma podemos solicitar a entrada de dados. Porém como vimos, cada dado é classificado por um tipo específico. No caso do valor de Porta, precisamos que ela seja do tipo “int”.

Verificando qual foi a classificação de dado atribuído ao “21” em nosso exemplo:

```
1 #!/usr/bin/python3
2 #Abaixo temos a funcao print() exibindo uma mensagem.
3 print("Learning Python - Desec Security\n")
4
5 porta = input("Informe uma porta: ")
6 print("\nPorta: ",porta,"Variavel do tipo:",type(porta))
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security

Informe uma porta: 21

Porta:  21 Variavel do tipo: <class 'str'>
```

E como vimos que o interpretador irá classificar automaticamente os dados conforme suas características, como esperado o valor “21” foi declarado como in… EPAA! **NÃO FOI identificado como int**!

Por quê??

Pois por padrão os valores coletados através da função input() **serão sempre do tipo *string*** (str).

Sabendo disso, **Podemos realizar a conversão do tipo de variáveis** (seguindo a lógica) a qualquer momento, inclusive durante a coleta com a função **input()**!

Convertendo Tipos de Variáveis

A conversão deverá seguir a regra da tabela explicada anteriormente, ou seja, para que o valor seja transformado ele precisa atender as características do tipo desejado!

Por exemplo, se uma variável do tipo ***string*** possui o valor de 21 armazenado, este valor tem os requisitos de ser do tipo **int**. Porém se o valor da string for “True”, “17.23...”, “abcde” e tentarmos a conversão para **int**, ocorrerá um erro pois tais variáveis não tem as características do tipo desejado a ser atribuído.

> **Recapitulando**: **QUALQUER VALOR** poderá assumir o tipo “string”, porém o contrário não se aplica. No sentido contrário, eles deverão suprir as características da classe.

Coletando os dados com a função **input()** definindo previamente o tipo de cada dado:

Tipos não informados previamente:

```
 1 #!/usr/bin/python3
 2 print("Learning Python - Desec Security\n")
 3
 4 porta = input("Informe uma porta: ")
 5 tempo = input("Informe um tempo em segundos: ")
 6 continuar = input("Deseja continuar? (True/False): ")
 7
 8 print("\nPorta: ",porta,"Variavel do tipo:",type(porta))
 9 print("Tempo: ",tempo,"Variavel do tipo:",type(tempo))
10 print("Continuar: ",continuar,"Variavel do tipo:",type(continuar))
~

Informe uma porta: 21
Informe um tempo em segundos: 17.21
Deseja continuar? (True/False): True

Porta:  21 Variavel do tipo: <class 'str'>
Tempo:  17.21 Variavel do tipo: <class 'str'>
Continuar:  True Variavel do tipo: <class 'str'>
```

Tipos declarados previamente:

```
 1 #!/usr/bin/python3
 2 print("Learning Python - Desec Security\n")
 3
 4 porta = int(input("Informe uma porta: "))
 5 tempo = float(input("Informe um tempo em segundos: "))
 6 continuar = bool(input("Deseja continuar? (True/False): "))
 7
 8 print("\nPorta: ",porta,"Variavel do tipo:",type(porta))
 9 print("Tempo: ",tempo,"Variavel do tipo:",type(tempo))
10 print("Continuar: ",continuar,"Variavel do tipo:",type(continuar))

root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security

Informe uma porta: 21
Informe um tempo em segundos: 17.21
Deseja continuar? (True/False): True

Porta:  21 Variavel do tipo: <class 'int'>
Tempo:  17.21 Variavel do tipo: <class 'float'>
Continuar:  True Variavel do tipo: <class 'bool'>
```

Ou podemos apenas coletar os dados e armazena-los do tipo string, e conforme a necessidade de manipulação da variável específica, podemos realizar a conversão de seu valor e armazenar em outra variável, invocando o tipo e a variável entre os parênteses da função:

```
 1 #!/usr/bin/python3
 2 print("Learning Python - Desec Security\n")
 3
 4 porta = input("Informe uma porta: ")
 5 tempo = input("Informe um tempo em segundos: ")
 6 continuar = input("Deseja continuar? (True/False): ")
 7
 8 print("\nPorta: ",porta,"Variavel do tipo:",type(int(porta)))
 9 print("Tempo: ",tempo,"Variavel do tipo:",type(float(tempo)))
10 print("Continuar: ",continuar,"Variavel do tipo:",type(bool(continuar)))
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security

Informe uma porta: 21
Informe um tempo em segundos: 17.13
Deseja continuar? (True/False): True

Porta:  21 Variavel do tipo: <class 'int'>
Tempo:  17.13 Variavel do tipo: <class 'float'>
Continuar:  True Variavel do tipo: <class 'bool'>
```

Independente do modo que for manipular os valores, o importante é passar o valor do tipo correto conforme a situação requisite.

> **Obs**: A conversão momentânea (em tempo de execução) não altera o tipo armazenado da variável, e sim apenas a saída ou utilização do valor no formado desejado **neste momento** (*string* para int por exemplo). Nestes casos, sempre que for utilizar uma variável de um tipo convertida em outro é necessário realizar a conversão.

Operadores Aritméticos

Em diversos cenários precisaremos realizar algum tipo de operação matemática para definir resultados, otimizar recursos e principalmente ao trabalharmos com comparações e decisões nas ferramentas criadas.

Os operadores matemáticos do Python seguem em suma a matemática elementar, tendo caracteres que identificam qual ação a ser tomada. As regras gerais da matemática aqui se aplicam, seguindo a ordem de execução da esquerda para a direita segundo as cláusulas dos parênteses.

A seguir, a tabela dos operadores matemáticos comuns:

Nome	**Operador**
Soma	+
Subtração	-
Multiplicação	*
Divisão	/

Tudo bem que a matemática pode parecer o básico, mas é importante saber que o Python permite manipularmos valores com operações reais (o que é utilizado para coisas simples até, como **laços de repetições** que veremos a seguir, ou envio de uma certa quantidade de bytes por exemplo…)

Exemplos básicos de operadores matemáticos e como o interpretador os executa:

```
1 #!/usr/bin/python3
2 print("Learning Python - Desec Security\n")
3
4 valor_inicial = 10
5 valor_final = 50
6 print("Soma: ",(valor_inicial + valor_final))
7 print("Subtracao:",(valor_final - valor_inicial))
8 print("Multiplicacao:",(valor_inicial * valor_final))
9 print("Divisao:",(valor_final / valor_inicial))
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security

Soma:  60
Subtracao: 40
Multiplicacao: 500
Divisao: 5.0
```

As regras da matemática também se aplicam no interpretador, porém as definições de o que será executado por primeiro, depois por segundo e assim sucessivamente é delimitado pelos parênteses (não é utilizado colchetes, chaves e etc como adicionais de hierarquia. É utilizado apenas parênteses).

Com isso podemos realizar a regra que for, os parênteses devem servir de escopo, onde primeiro é executado algo. Do contrário, o cálculo será interpretado da esquerda para a direita seguindo a regra hierárquica dos sinais matemáticos.

```
1 #!/usr/bin/python3
2 print("Learning Python - Desec Security\n")
3
4 valor_inicial = 10
5 valor_final = 50
6 print("Calculo simples: ",(valor_inicial + valor_final / valor_inicial))
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security

Calculo simples:  15.0
```

O calculo realizado foi: 10 + 50 / 10

Primeiro foi interpretado o 50/10 = 5 e posteriormente a adição, 10+5, que retornou 15.

Caso fosse preciso que a adição fosse interpretada antes da divisão, deve ser inserido os parênteses para limitar a hierarquia:

```
1 #!/usr/bin/python3
2 print("Learning Python - Desec Security\n")
3
4 valor_inicial = 10
5 valor_final = 50
6 print("Calculo simples: ",((valor_inicial + valor_final) / valor_inicial))
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security

Calculo simples:  6.0
```

Dessa forma o cálculo é realizado: 10+50 = 60 / 10 = **6**

Pode ser inserido quantos parêntesis for necessário, um dentro do outro, limitando assim várias hierarquias e ordem a ser executado o cálculo. Isso depende tudo da lógica a ser aplicada.

Bom, mas focando nos scripts, algo legal que pode ser feito é utilizar a soma para manipular strings!

Há muitas formas de manipular strings com cálculos, alguns exemplos mais simples são estes:

```
1 #!/usr/bin/python3
2 print("Learning Python - Desec Security\n")
3
4 nome = "DESEC"
5 print("DESEC x3", nome * 3)
6 print("Caractere A x10:","A" * 10)
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security

DESEC x3 DESECDESECDESEC
Caractere A x10: AAAAAAAAAA
```

Podemos também utilizar os métodos da classe string para realizar manipulações desejadas. Para verificar os métodos possíveis basta utilizar o IDLE com esta sintaxe:

>>> help(str)

Dessa forma é exibido todas os métodos permitidos pela classe:

```
Help on class str in module builtins:

class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
 |  Create a new string object from the given object. If encoding or
 |  errors is specified, then the object must expose a data buffer
 |  that will be decoded using the given encoding and error handler.
 |  Otherwise, returns the result of object.__str__() (if defined)
 |  or repr(object).
 |  encoding defaults to sys.getdefaultencoding().
 |  errors defaults to 'strict'.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __format__(self, format_spec, /)
 |      Return a formatted version of the string as described by format_spec.
 |
:
```

OPERADORES DE COMPARAÇÃO

A tomada de decisão em programação é um ponto muito importante, ainda mais tratando-se de ferramentas ofensivas, pois será através da resposta obtida que o programa "decidirá" qual ação a ser tomada.

As decisões são tomadas mediante a expressão a ser avaliada e o que deverá ocorrer. Caso a condição seja verdadeira ocorrerá algo, caso falso também.

Em python, se a condição for verdadeira o interpretador irá identificar como "**is true**" (é verdadeira), caso seja falsa será identificada como "**is not true**" (não é verdadeira).

Para tal, a estrutura seria basicamente dessa forma:

if (expressão):
bloco_de_instrução_caso_verdadeiro
else:
bloco_de_instrução_caso_falso

IF (se): Tem a função de verificar se a expressão a seguir é verdadeira.

ELSE (então): É um complemento para o IF, caso a expressão não seja verdadeira as instruções.

Abaixo temos uma tabela dos operadores relacionais para melhor compreensão. Utilizaremos duas variaveis para exemplo:

TABELA DOS OPERADORES RELACIONAIS			
NOME (descrição)	**OPERADOR**	**DESCRIÇÃO**	**EXEMPLO**
Igual a...	Igual	Se o valor dos dois operadores for iguais, a condição é verdadeira.	(var_1 == var_2) = **is not true**

Maior que...	>	Se o valor do operador da esquerda for maior que o da direita, a condição é verdadeira.	(var_1 > var_2) = **is not true**
Menor que…	<	Se o valor do operador da direita for maior que o da direita, a condição é verdadeira.	(var_2 < var_1) = **is not true**
Diferente de...	!=	Se o valor dos dois operadores for diferente, a condição é verdadeira.	(var_1 != var_2) = **is true**
Maior ou igual a...	>=	Se o valor do operador da esquerda for maior ou igual ao da direita, a condição é verdadeira.	(var_1 >= var_2) = **is not true**
Menor ou igual a...	<=	Se o valor do operador da esquerda for menor ou igual ao da direita, a condição é verdadeira.	(var_1 <= var_2) = **is true**

Temos ainda o operador “<>” (diferente de…), porém foi removido do Python3 sendo aconselhado utilizar sempre o “!=” como operador da condição “**diferente de...**”.

A seguir alguns exemplos de comparação simples, vamos verificar se uma porta é maior que outra:

```
1 #!/usr/bin/python3
2 print("Learning Python - Desec Security\n")
3
4 porta_1 = 21
5 porta_2 = 80
6
7 if (porta_1 > porta_2):
8     print("A porta",porta_1,"eh maior que a porta",porta_2)
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security

root@desecsecurity:~/python#
```

Não foi exibido nada, pois a condição é falsa. A “porta_1” não é maior que a “porta_2”, porém a tomada de decisão não tem instrução do que fazer quando a condição é negativa neste caso. Podemos então utilizar o **else** para melhorar a decisão:

```
1 #!/usr/bin/python3
2 print("Learning Python - Desec Security\n")
3
4 porta_1 = 21
5 porta_2 = 80
6
7 if (porta_1 > porta_2):
8     print("A porta",porta_1,"eh maior que a porta",porta_2)
9 else:
10     print("A porta",porta_1,"nao eh maior que a porta",porta_2)
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security

A porta 21 nao eh maior que a porta 80
```

Agora a resposta foi satisfatória. Neste caso como a primeira condição não era verdadeira a instrução abaixo não foi executada. Como neste caso havia o Else ele passa a ser o “caso real” da comparação e é executado.

Podemos também operar com palavras, comparando strings!

```
#!/usr/bin/python3
print("Learning Python - Desec Security\n")

usuario = input("Insira um usuario: ")

if (usuario == 'admin'):
    print("Tentativa de login com usuario Admin")
else:
    print("Usuario desconhecido")
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security

Insira um usuario: administrator
Usuario desconhecido
```

Porém, ao passarmos Admin o script reconhece como verdade a comparação e executa o print, no código abaixo:

```
#!/usr/bin/python3
print("Learning Python - Desec Security\n")

usuario = input("Insira um usuario: ")

if (usuario == 'admin'):
    print("Tentativa de login com usuario Admin")
else:
    print("Usuario desconhecido")
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security

Insira um usuario: admin
Tentativa de login com usuario Admin
```

Podemos então colocar uma outra condição caso o usuário senha conhecido, a requisição de uma senha:

```
#!/usr/bin/python3
print("Learning Python - Desec Security\n")

usuario = input("Insira um usuario: ")

if (usuario == 'admin'):
    print("Tentativa de login com usuario Admin..\n")
    senha = input("Digite a senha do admin: ")
    if (senha == 'admin'):
        print("Login efetuado com sucesso!")
    else:
        print("Senha incorreta.")
else:
    print("Usuario desconhecido")
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security

Insira um usuario: admin
Tentativa de login com usuario Admin..

Digite a senha do admin: 123
Senha incorreta.
```

A última chamada foi para o print de Senha Incorreta. O script não percorreu e executou o último print pois a condição de usuário ser igual a ‘admin’ foi verdadeira, com isso ele entra neste escopo e encerra por ali.

Caso a senha tivesse sido correta, a mensagem de sucesso teria sido exibida:

```
#!/usr/bin/python3
print("Learning Python - Desec Security\n")

usuario = input("Insira um usuario: ")

if (usuario == 'admin'):
    print("Tentativa de login com usuario Admin..\n")
    senha = input("Digite a senha do admin: ")
    if (senha == 'admin'):
        print("Login efetuado com sucesso!")
    else:
        print("Senha incorreta.")
else:
    print("Usuario desconhecido")
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security

Insira um usuario: admin
Tentativa de login com usuario Admin..

Digite a senha do admin: admin
Login efetuado com sucesso!
```

Além do IF e ELSE, temos o **ELIF**!

O ELIF entra no meio das condições quando inúmeras possibilidades podem ser testadas, por exemplo:

if (expressão):
bloco_de_instrução_caso_verdadeiro
elif (expressão):
bloco_de_instrução_caso_falso
elif (expressão):
bloco_de_instrução_caso_as_anteriores_sejam_falso
eles:
bloco_de_instrução_caso_falso (caso nenhuma das anteriores tenha atendido)

Igualdade:

```
#!/usr/bin/python3
print("Learning Python - Desec Security\n")

porta_1 = int(input("Insira uma porta: "))
porta_2 = int(input("Insira uma segunda porta: "))

if (porta_1 == porta_2):
    print("Portas iguais.")
elif (porta_1 < porta_2):
    print("A porta",porta_1,"eh menor que a porta",porta_2)
else:
    print("A porta",porta_1,"eh maior que a porta",porta_2)
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security

Insira uma porta: 21
Insira uma segunda porta: 21
Portas iguais.
```

Menor que:

```
 1 #!/usr/bin/python3
 2 print("Learning Python - Desec Security\n")
 3
 4 porta_1 = int(input("Insira uma porta: "))
 5 porta_2 = int(input("Insira uma segunda porta: "))
 6
 7 if (porta_1 == porta_2):
 8     print("Portas iguais.")
 9 elif (porta_1 < porta_2):
10     print("A porta",porta_1,"eh menor que a porta",porta_2)
11 else:
12     print("A porta",porta_1,"eh maior que a porta",porta_2)
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security

Insira uma porta: 21
Insira uma segunda porta: 80
A porta 21 eh menor que a porta 80
```

Maior que:

```
 1 #!/usr/bin/python3
 2 print("Learning Python - Desec Security\n")
 3
 4 porta_1 = int(input("Insira uma porta: "))
 5 porta_2 = int(input("Insira uma segunda porta: "))
 6
 7 if (porta_1 == porta_2):
 8     print("Portas iguais.")
 9 elif (porta_1 < porta_2):
10     print("A porta",porta_1,"eh menor que a porta",porta_2)
11 else:
12     print("A porta",porta_1,"eh maior que a porta",porta_2)
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security

Insira uma porta: 2121
Insira uma segunda porta: 443
A porta 2121 eh maior que a porta 443
```

Podemos ainda utilizar múltiplas comparações em uma condição, onde a comparação pode ser:

if comparacao **OU** outra_comparação = **or**
if comparacao **E** outra_comparacao = **and**

```
 1 #!/usr/bin/python3
 2 print("Learning Python - Desec Security\n")
 3
 4 porta_1 = int(input("Insira uma porta: "))
 5 porta_2 = int(input("Insira uma segunda porta: "))
 6
 7 if (porta_1 < porta_2 and porta_1 > 21):
 8     print("A porta",porta_1,"eh menor que a porta",porta_2,"e eh maior que 22")
 9 else:
10     print("Condicao nao bateu.")
11
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security

Insira uma porta: 80
Insira uma segunda porta: 2121
A porta 80 eh menor que a porta 2121 e eh maior que 22
```

Ao utilizarmos a condição “and” (E), ambas as condições precisam ser verdadeiras para entrar no bloco de instrução abaixo. Caso uma delas seja verdadeira e a outra falsa, a comparação toda estará sendo declarada falsa...

Afinal estamos exigindo: “Só entra nessa condição se a porta 1 for menor que a porta 2 E maior que 21.. senão não...”:

```
#!/usr/bin/python3
print("Learning Python - Desec Security\n")

porta_1 = int(input("Insira uma porta: "))
porta_2 = int(input("Insira uma segunda porta: "))

if (porta_1 < porta_2 and porta_1 > 21):
    print("A porta",porta_1,"eh menor que a porta",porta_2,"e eh maior que 22")
else:
    print("Condicao nao bateu.")
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security

Insira uma porta: 21
Insira uma segunda porta: 80
Condicao nao bateu.
```

As tomadas de decisões podem ser ainda mais complexas, envolvendo vários **elifs** e condições adicionais, tudo irá depender da lógica necessária a ser aplicada para cada caso.

LAÇO DE REPETIÇÃO

For Loop

O laço de repetição “for” é utilizado para executarmos repetidamente as mesmas instruções durante um intervalo definido. Para utilizarmos o laço FOR, precisamos definir uma variável que irá armazenar a contagem. A função “range” servirá para definir a quantidade de vezes que as instruções desejadas sejam executadas.

Estrutura exemplo:

for i in range(0,11):
bloco_de_código

```
for i in range(0,11): #Criamos a variavel Indice com o nome i, limitando a execucao em 11x.
        print("Executando o loop na posicao",i) #Instrucao a ser executada 10x conforme o indice definido.

```

```
root@desecsecurity:~/python# python script.py
('Executando o loop na posicao', 0)
('Executando o loop na posicao', 1)
('Executando o loop na posicao', 2)
('Executando o loop na posicao', 3)
('Executando o loop na posicao', 4)
('Executando o loop na posicao', 5)
('Executando o loop na posicao', 6)
('Executando o loop na posicao', 7)
('Executando o loop na posicao', 8)
('Executando o loop na posicao', 9)
('Executando o loop na posicao', 10)
```

Por padrão, bom funcionamento, e boas práticas, o índice para laços de repetição, ou posições de armazenamento (que veremos adiante) deverá sempre iniciar em Zero, onde ao atingir o limite do range a função irá parar.

A definição da função range para nosso exemplo seria algo como:

“Começando a contagem em zero, executar as instruções a seguir 11x, ou seja, de 0 a 10. Quando o índice atingir a posição 11 pare, do contrário estará executando 12x”.

É possível definir um objetivo para o laço FOR. Digamos que queremos executar as instruções em um determinado range, porém SE atingir algum valor o laço FOR poderá parar as execuções e o script seguir o fluxo adiante:

```
1 for i in range(0,11):
2         print("Executando o loop na posicao",i)
3         if i == 7:
4                 print("Atingido a posicao 7. Parando a execucao.")
5                 break
6
```

```
root@desecsecurity:~/python# python script.py
('Executando o loop na posicao', 0)
('Executando o loop na posicao', 1)
('Executando o loop na posicao', 2)
('Executando o loop na posicao', 3)
('Executando o loop na posicao', 4)
('Executando o loop na posicao', 5)
('Executando o loop na posicao', 6)
('Executando o loop na posicao', 7)
Atingido a posicao 7. Parando a execucao.
```

Vemos no exemplo acima que ainda que definido para o **for** ser executado até a posição 11 havia o IF, limitando sua parada ao identificar o valor correspondente. A instrução **break** foi utilizada para o programa parar.

Break: A instrução break tem como finalidade finalizar a iteração e passar a execução do script para o próximo bloco de código.

O processo do exemplo acima pode não parecer fazer sentido prático visto agora na usabilidade, porém torna-se muito útil quando trabalhado com buscas através de uma estrutura de Listas (veremos adiante) por exemplo onde normalmente é preciso tomar decisões para execução de ações ao encontrar um resultado desejado.

WHILE

Assim como o laço de repetição com o FOR, o While atua de forma muito similar na repetição da instrução, porém diferente do **for** que atua com um índice inicial e final (tornando muitas vezes sua execução no fluxo obrigatória), o while atuará apenas enquanto a condição definida seja verdadeira, com isso caso a condição ao entrar no while já seja falsa o while não entrará em execução.

Exemplo do fluxo suprindo a condição verdadeira:

```
root@desecsecurity:~/python# python script.py
Codigo sendo executado...
Digite C para Continuar ou E para terminar a execucao: C
Codigo sendo executado...
Digite C para Continuar ou E para terminar a execucao: C
Codigo sendo executado...
Digite C para Continuar ou E para terminar a execucao: C
Codigo sendo executado...
Digite C para Continuar ou E para terminar a execucao: E
Finalizando o processo
```

```
1 resposta = str('C')
2 while (resposta == 'C'):
3         print("Codigo sendo executado...")
4         resposta = raw_input("Digite C para Continuar ou E para terminar a execucao: ")
5 print("Finalizando o processo")
6
```

Como dito, o código armazena uma resposta e tomará uma ação enquanto ela for verdadeira. Iniciou-se com ela sendo “C” suprindo a requisição. O loop assim é executado indeterminadamente, até que a condição seja falsa (quando o usuário digitar E).

Caso a condição já entre como E na checagem, o while não é executado:

```
1 resposta = str('E')
2 while (resposta == 'C'):
3         print("Codigo sendo executado...")
4         resposta = raw_input("Digite C para Continuar ou E para terminar a execucao: ")
5 print("Finalizando o processo")
6
```

```
root@desecsecurity:~/python# python script.py
Finalizando o processo
```

O QUE FOI VISTO ATÉ AGORA?

Identação: Vimos que um dos recursos mais essenciais do Python é a identação onde organiza corretamente o escopo do código segundo os espaçamentos. Com ela delimitamos corretamente os escopos dos blocos de instrução.

Blocos de Instruções: Nem sempre será executado apenas um comando, muitas vezes será necessário executar uma sequência de instruções no mesmo nível hierárquico onde o interpretador realizará segundo a lógica de cima para baixo, da esquerda para a direita as instruções propostas, até chegar no final e tomar alguma outra ação (ou finalizar).

Execução dos scripts Python: Podemos utilizar arquivos .py para criar scripts e invocar o interpretador para executa-los, ou simplesmente utilizar o IDLE do Python para testar rápidos recursos sem a necessidade de criação de um arquivo.

Inputs e Outputs: Verificamos também como funcionam as saídas de dados na tela para mensagens intuitivas ou requisição de dados para o usuário, comentários no código, e também a entrada de dados através dos inputs.

Variáveis: Como precisamos trabalhar com dados precisamos armazená-los em algum lugar, então utilizamos as variáveis! Cada tipo de dado terá uma característica, e devemos instruir o tipo correto de cada um (ou deixar o Python escolher segundo suas características).

Até então já vimos o básico para entender a estruturação dos dados em Python. Para o Port Scanner precisamos de algumas informações adicionais para que o script fique mais completo.

Podemos por exemplo armazenar valores em uma variável, porém e se quisermos salvar vários valores? Por exemplo 10... criamos 10 variáveis? É mais elegante não, e sim usar alguma estruturação de dados.

Temos a seguir o conceito de listas que consiste em “uma variável” com vários espaços internos, onde neles podem ser armazenados elementos (valores) para evitarmos criar várias variáveis do mesmo tipo para algum propósito.

ESTRUTURAS DE DADOS

O objetivo das estruturas de dados em Python é armazenar um conjunto de informações, normalmente do mesmo tipo. São similares as “variáveis” (objetos), porém diferente das variáveis que armazenam apenas um valor, as estruturas de dados armazenam vários valores, os quais são acessíveis através de um índice.

Alguma das estruturas de dados mais utilizadas em Python são:

- Listas
- Dicionário
- Tuplas (será visto na prática, com o socket)

Listas

Uma lista armazena um conjunto de valores onde são chamados de elementos e cada elemento é identificado por um **índice único**. Na lista cada elemento poderá ser de um tipo, porém normalmente como ela é utilizada para um propósito específico é comum que os elementos sejam da mesma classe.

A forma mais simples de criar uma lista é definindo uma sequência de valores entre colchetes “[ ]” e separados por vírgula.

Exemplo de uma lista vazia:

nome_da_lista = []

```
>>> lista = []
>>> print(lista)
[]
```

Com a criação de uma lista vazia, os valores poderão ser inseridos durante a execução do programa sendo coletados, ou manualmente. O novo valor inserido em uma lista sempre entrará no final por padrão, criando um indice a mais consequentemente.

Há a possibilidade de criarmos uma lista de **portas** com valores definidos, por exemplo:

portas = [21, 22, 53, 80]

Temos então uma lista com quatro elementos. O índice de cada elemento começará sempre em 0, onde os próximos elementos sempre seguirão a sequência. Neste caso temos os seguintes indices:

portas[]	
Índice	**Elemento**
0	21
1	22
2	53
3	80

Acessando os elementos

Para acessarmos o elemento correspondente a algum índice usamos os colchetes ‘[]’ onde é informado o índice desejado:

print**(**nome_da_lista**[numero_do_indice]**)

```
#!/usr/bin/python3
print("Learning Python - Desec Security\n")

portas = [21, 22, 53, 80]
print(portas[2])
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security

53
```

Ao informarmos que queremos o valor no índice 2, o **elemento** 53 é exibido.

Quando é requisitado um valor de índice que não existe na lista, é informado um erro: "índice de lista fora do range" (*list index out of range*), ou seja, o índice não existe:

```
#!/usr/bin/python3
print("Learning Python - Desec Security\n")

portas = [21, 22, 53, 80]
print(portas[5])
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security

Traceback (most recent call last):
  File "./script.py", line 5, in <module>
    print(portas[5])
IndexError: list index out of range
```

Um outro ponto interessante é que podemos requisitar um índice negativo. Os índices negativos percorrem o valor da lista de trás para frente, onde o último elemento corresponde ao índice "-1":

```
#!/usr/bin/python3
print("Learning Python - Desec Security\n")

portas = [21, 22, 53, 80]
print("Ultimo elemento da lista:",portas[-1])
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security

Ultimo elemento da lista: 80
```

Claro que esta regra se aplica ao limite máximo, ao requisitarmos um índice negagivo que sai do range existente, como o "-5" em diante, será exibido o mesmo erro de que estamos requisitando um índice fora do alcance (que não existe na lista):

```
#!/usr/bin/python3
print("Learning Python - Desec Security\n")

portas = [21, 22, 53, 80]
print("Index alem do limite:",portas[-5])
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security

Traceback (most recent call last):
  File "./script.py", line 5, in <module>
    print("Index alem do limite:",portas[-5])
IndexError: list index out of range
```

É possível também requisitar **todos os elementos** que a lista contém através do *print*:

```
1 #!/usr/bin/python3
2 print("Learning Python - Desec Security\n")
3
4 portas = [21, 22, 53, 80]
5 print("Lista completa:",portas)
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security

Lista completa: [21, 22, 53, 80]
```

Podemos ainda informar os índices que queremos acessar em forma de **intervalo**, assim é possível verificar vários elementos de uma única vez.

Fatiamento de Lista: O fatiamento ocorre com o delimitador ":" onde é definido um intervalo a ser exibido.

Por exemplo:

nome_da_lista[**numero_indice:numero_indice**]

Neste caso, será exibido o intervalo entre o índice 1 e 3:

```
1 #!/usr/bin/python3
2 print("Learning Python - Desec Security\n")
3
4 portas = [21, 22, 53, 80]
5 print("Fatiamento da lista:",portas[1:3])
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security

Fatiamento da lista: [22, 53]
```

O índice 1 é o limite e por isso ele é exibido, já o 3 é o limitador! O limitador não é exibido, pois quando chega nele o processo para. Ou seja, entre o range "1:3" é exibido apenas o elemento na posição "1" e "2". Lembrando que a Lista se inicia pelo índice zero, e o 21 é o elemento que o contempla.

O intervalo pode ser definido do índice 0 ao range desejado, utilizando apenas dois pontos antes do limitador, ficando da seguinte forma:

```
1 #!/usr/bin/python3
2 print("Learning Python - Desec Security\n")
3
4 portas = [21, 22, 53, 80]
5 print("Fatiamento da lista:",portas[:2])
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security

Fatiamento da lista: [21, 22]
```

Da mesma forma o índice 2 (onde está o elemento 53) limita a quebra, onde ele não é exibido. Ao utilizarmos o fatiamento de listas, o índice inicial quando não informado como no exemplo acima será sempre zero, assim como o delimitador, onde **portas[:]** exibiria o resultado de todos os elementos da lista.

> ***Dica**: O fatiamento de listas pode ser útil ao identificar uma lista muito grande e desejarmos criar regras para **percorrer os elementos** segundo alguma lógica necessária. (Veremos a seguir como percorrer os elementos de uma lista).*

Adicionando elementos em uma posição da lista
Podemos adicionar elementos em um intervalo que desejamos utilizando um range entre o número do índice.

Abaixo, faremos a inserção de um elemento no índice 3 (sem alterar os elementos originais da lista), e também 3 novos elementos entre o elemento 3 e 4:

```
#!/usr/bin/python3
print("Learning Python - Desec Security\n")

portas = [21, 22, 53, 80]
print("Lista original:",portas)
# Adicionando um elemento no indice 3
portas[3:3] = [111]
print("Lista alterada:",portas)
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security

Lista original: [21, 22, 53, 80]
Lista alterada: [21, 22, 53, 111, 80]
```

A inserção foi feita com sucesso. A lista atualiza os índices automaticamente sempre que manipulada.

Seguindo esta lógica podemos adicionar mais de um elemento na lista em alguma posição específica, basta separá-los por vírgula:

```
#!/usr/bin/python3
print("Learning Python - Desec Security\n")

portas = [21, 22, 53, 80]
print("Lista original:",portas)
# Adicionando um elemento no indice 3
portas[3:3] = [111, 8080, 443]
print("Lista alterada:",portas)
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security

Lista original: [21, 22, 53, 80]
Lista alterada: [21, 22, 53, 111, 8080, 443, 80]
```

Substituindo elementos
Podemos adicionar novos elementos às posições já existente. Para isso, basta informar o índice com o novo valor:

```
#!/usr/bin/python3
print("Learning Python - Desec Security\n")

portas = [21, 22, 53, 80]
print("Lista original:",portas[:])
# Adicionando um elemento no indice 3
portas[1] = 443
print("Lista alterada:",portas[:])
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security

Lista original: [21, 22, 53, 80]
Lista alterada: [21, 443, 53, 80]
```

A substituição poderá ser feita em massa da mesma forma que utilizamos o delimitador ":" para exibir o range. Para isso a sintaxe neste exemplo é a seguinte:

lista[X:X] = [valor, valor, valor]

```
#!/usr/bin/python3
print("Learning Python - Desec Security\n")

portas = [21, 22, 53, 80]
print("Lista original:",portas[:])
# Adicionando um elemento no indice 3
portas[0:3] = 443, 8080, 2121
print("Lista alterada:",portas[:])
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security

Lista original: [21, 22, 53, 80]
Lista alterada: [443, 8080, 2121, 80]
```

No exemplo acima, realizamos a substituição dos elementos correspondentes aos índices 0, 1 e 2.

Remoção de elementos da lista
O modo mais fácil de remoção de itens de uma lista é com a utilização de métodos!

Hm, mas o que são métodos?

Bem a lista é uma classe, um tipo de "variável" assim como outras, e também possui suas particularidades e métodos que a apoiam.

Para entender quais funcionalidades uma classe tem, podemos utilizar o help(), o qual funciona similar ao "man".

Exemplo:

>>> help(classe)

Dessa forma é informado uma saída de todas as funcionalidades possíveis para diferentes classes no Python, neste caso veremos a lista:

>>> help(list)

```
Help on class list in module builtins:

class list(object)
 |  list(iterable=(), /)
 |
 |  Built-in mutable sequence.
 |
 |  If no argument is given, the constructor creates a new empty list.
 |  The argument must be an iterable if specified.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __delitem__(self, key, /)
 |      Delete self[key].
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
:
```

Podemo ir pressionando enter até chegar ao final para verificar as funcionalidades completas.

Encontramos o método delete:

```
 |
 |  __delitem__(self, key, /)
 |      Delete self[key].
 |
```

A sintaxe para utilizarmos o remove como visto é a seguinte:

del lista[indice]

```
 1 #!/usr/bin/python3
 2 print("Learning Python - Desec Security\n")
 3
 4 portas = [21, 22, 53, 80]
 5 print("Lista original:",portas[:])
 6 # Adicionando um elemento no indice 3
 7 del portas[0]
 8 print("Lista alterada:",portas[:])
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security

Lista original: [21, 22, 53, 80]
Lista alterada: [22, 53, 80]
```

Existem diferentes métodos para ser utilizados em diferentes funções que queremos. Sempre que houver dúvidas é recomendado dar o comando **help** seguido do que deseja obter maiores detalhes!

O mesmo pode ser aplicado para intervalos como visto anteriormente:

```
1 #!/usr/bin/python3
2 print("Learning Python - Desec Security\n")
3
4 portas = [21, 22, 53, 80]
5 print("Lista original:",portas[:])
6 # Adicionando um elemento no indice 3
7 del portas[1:4]
8 print("Lista alterada:",portas[:])
```

```
root@desecsecurity:~/python# ./script.py
Learning Python - Desec Security

Lista original: [21, 22, 53, 80]
Lista alterada: [21]
```

DANDO VIDA AO PORT SCANNER

Até agora o que fizemos foi ver conceitos básicos sobre lógica de programação, unificando-os aos recursos que o Python oferece. Com o Python contamos com inúmeras classes e diferentes métodos que facilitam o processo de criação de ferramentas, possibilitando economizar um bom tempo e desempenho na realização dos scripts.

Os conceitos visto até agora já formaram uma boa base para você conseguir **ler e escrever em Python**!

O material disponibilizado não te tornou um mestre na programação e nem te apresentou todos os potenciais recursos da linguagem, mas serviu como a base para os principais recursos disponíveis dando-lhe o caminho para trilhar os próximos passos!

A seguir vamos começar a desenvolver o script do Port Scanner utilizando todos os recursos vistos anteriormente, de uma forma (mais complexa e sem a utilização de tantos recursos), mas que possibilite treinar ainda mais e ver aplicado tudo o que foi visto (e mais um pouco)!

HANDS ON! [Port Scanner]

Um PortScanner é um script criado para verificar as portas abertas em um determinado host. Como você já acompanhou os módulos anteriores já tem familiaridade com o termo e inclusive com o script, com isso vamos passar adiante as explicações a respeito dele e partir em como podemos desenvolvê-lo utilizando os recursos até agora vistos.

Estrutura do Port Scanner

A seguir temos uma estrutura básica de um Port Scanner

```
1 #!/bin/python3
2 import socket
3 ip = '37.59.174.225'
4 for porta in range(1,1024):
5     sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
6     if (sock.connect_ex((ip,porta)) == 0):
7         print("[+] Porta TCP Aberta:",porta)
8         sock.close()
```

Para entender melhor cada etapa, vamos para uma breve descrição de cada linha:

```
1 #!/bin/python3
```

Aqui como já é conhecido, é a primeira linha (não obrigatória) onde referenciamos onde o interpretador Python está instalado em nossa máquina. Esta é uma linha opcional, pois pode ser interpretado o Script direto pelo interpretador: **python3 script.py**

```
2 import socket
```

Os **imports** serão algo que veremos a seguir. Através dessa funcionalidade é possível importar módulos e utilizar os recursos que as classes disponibilizam. Os sockets possuem a funcionalidade de

enviar e receber dados, e para tal esta utilização no port scanner é fundamental para identificarmos se uma porta está aberta ou fechada referente às informações que obtemos de resposta.

```
3 ip = '37.59.174.225'
```

A variável “ip” neste caso está armazenando o IP do alvo onde o socket irá conectar-se.

```
4 for porta in range(1,1024):
```

Nesta etapa está sendo criado um laço de 1024 repetições onde cada execução irá se conectar em uma porta, iniciando na 1, e terminando na 1024.

```
5     sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Aqui estamos criando uma referência chamada “sock” (pode ser qualquer nome, apenas deixei intuitivo para referenciar a classe socket). Estamos informando qual é o **tipo do socket** que será utilizado. No caso o SOCK_STREAM é o **socket de fluxo** utilizado para realizar conexões **TCP**, caso fosse realizar um portscan em portas **UDP** (que não é o foco no momento) o tipo de socket seria o **SOCK_DGRAM**, sockets de datagrama.

```
6     if (sock.connect_ex((ip,porta)) == 0):
```

Esta é a tomada de decisão onde é verificado se a porta está realmente aberta, ou fechada.
Acima estamos informando para o *socket* se conectar em um IP seguido de uma PORTA. É uma decisão simples exemplificada, porém na prática é algo mais complexo pois uma porta pode se comportar de diferentes maneiras e ainda assim estar aberta dependendo das configurações empregadas no *webserver*, nestes casos um simples Port Scanner não poderá validar com certeza o estado da porta. Partindo do princípio básico acima, a condição “== 0” significa que não houve erros para se conectar, então a porta “está aberta”.

```
7         print("[+] Porta TCP Aberta:",porta)
```

Após a conexão ser realizada e não retornando erro, estamos enviando uma mensagem para a tela de que a porta está aberta!

```
8         sock.close()
```

E em seguida fechamos a conexão com a classe .close(). Esta parte é mais por boas práticas, pois ao final o socket fecha sua conexão normalmente. Caso ocorra algum erro que a conexão permaneça ativa, a função fará com que ele feche para então dar início ao looping seguinte do For configurado na quarta linha.

Lembra que anteriormente falamos sobre listas, dicionários e Tupas? Bem, não foi explicado o que é uma tupla pois teriamos um exemplo aqui!
Dicionários e listas são mutaveis ou seja, podem sofrer modificações. Tuplas não, tuplas são imutáveis. A grande diferença entre as duas é que as listas e dicionários (de um ponto de vista mais conceitual), devem no geral ser estruturadas para armazenar valores do mesmo tipo. Já as tuplas (*tuples*) são conceituadas para armazenar dados diferentes, como a seguinte:

Na linha 6 temos o exemplo de uma tupla, onde é composta por “ip + porta”.

```
6     if (sock.connect_ex((ip,porta)) == 0):
```

Mas por que há dois parênteses para abrir e dois para fechar? Vamos mudar e deixar apenas com um?

```
  6     if (sock.connect_ex(ip,porta) == 0):
```

Executando o script:

```
root@desecsecurity:~/python# ./portscan.py
Traceback (most recent call last):
  File "./portscan.py", line 6, in <module>
    if (sock.connect_ex(ip,porta) == 0):
TypeError: connect_ex() takes exactly one argument (2 given)
```

Recebemos um erro que informa: connect_ex() pega exatamente **UM** argumento (2 foram dados).

Isso ocorre por que a Tupla **((**IP,PORTA**))** é considerado apenas um argumento, um conjunto digamos assim, já **(**IP,PORTA**)** é considerado dois argumentos pois está informando duas variáveis. Em resumo, a tupla armazena todas as informações como sendo uma só.

Perfeito! Até aqui você já tem uma boa noção de como o socket opera e quais os passos ele segue para realizar a conexão e validação da porta aberta/fechada.

Vamos agora identificar quanto tempo demora aproximadamente para um scan utilizando o nosso script atual. Para isso vamos realizar algumas implementações nele:

```
 1 #!/bin/python3
 2 import socket
 3 from datetime import datetime
 4
 5 ip = '37.59.174.225'
 6
 7 inicio = datetime.now()
 8
 9 for porta in range(1,100):
10     sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
11     if (sock.connect_ex((ip,porta)) == 0):
12         print("[+] Porta TCP Aberta:",porta)
13         sock.close()
14 final = datetime.now()
15 tempo_total = (final - inicio)
16 print("\nO tempo de duracao do scan foi de:",tempo_total)
```

Realizamos acima o import o *datetime* para coletar o horário do sistema. Abaixo teremos duas novas variáveis declaradas: o tempo **inicial**, e o tempo **final**. No final, a variável **tempo_total** irá nos informar quanto tempo levou para o script realizar o escaneamento.

Para ser algo mais rápido (visto que este irá demorar um pouco), limitei o range de portas de 1 a 100, escaneando assim as 100 primeiras portas do webserver.

Em execução:

```
root@desecsecurity:~/python# ./portscan.py
[+] Porta TCP Aberta: 21
[+] Porta TCP Aberta: 22
[+] Porta TCP Aberta: 53
[+] Porta TCP Aberta: 80

O tempo de duracao do scan foi de: 0:06:26.929103
```

O tempo de execução para as 100 portas foi de 6 minutos e 26s. Algo em torno de 3,6 segundos cada conexão.

Podemos implementar um *timeout* para o intervalo de uma requisição em outra em nosso socket. Para isso podemos realizar o procedimento informando que o timeout padrão será de 0.5 segundos por exemplo, menos que isso pode ocorrer que nossas requisições não sejam processadas corretamente.

```
 1 #!/bin/python3
 2 import socket
 3 from datetime import datetime
 4
 5 ip = '37.59.174.225'
 6
 7 inicio = datetime.now()
 8
 9 for porta in range(1,100):
10     sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
11
12     socket.setdefaulttimeout(0.5)
13
14     if (sock.connect_ex((ip,porta)) == 0):
15         print("[+] Porta TCP Aberta:",porta)
16         sock.close()
17 final = datetime.now()
18 tempo_total = (final - inicio)
19 print("\nO tempo de duracao do scan foi de:",tempo_total)
```

Na linha 12 acima declaramos o timeout das requisições para 0.5 segundos, com isso podemos verificar o tempo de execução do Port Scanner nas 100 portas a seguir:

```
root@desecsecurity:~/python# ./portscan.py
[+] Porta TCP Aberta: 21
[+] Porta TCP Aberta: 22
[+] Porta TCP Aberta: 53
[+] Porta TCP Aberta: 80

O tempo de duracao do scan foi de: 0:00:51.727049
```

Dessa vez o tempo caiu consideravelmente, terminando o processo em 51.7 segundos.

Este processo ainda não é rápido se levarmos em conta que existem 65535 portas no webserver para serem escaneadas, porém para fins didáticos pode ser compreendido de uma forma simples como é realizado.

Existem formas que otimizam o port scanner e melhoram o seu processamento, levando em conta o tipo de conexão que é realizado, a utilização de *threadins* entre outros fatores. Um exemplo pode ser visto a seguir, sobre tempo de execução das 65535 portas em um script que desenvolvemos:

```
root@desecsecurity:~/scriptsDS# python3 portScannerDS.py
IP: 37.59.174.225
Escaneando o IP: [ 37.59.174.225 ]
[+] Porta Aberta: 21
[+] Porta Aberta: 22
[+] Porta Aberta: 53
[+] Porta Aberta: 80
[+] Porta Aberta: 111
Tempo de execucao:  182.84
```

Neste caso é possível identificar que há recursos disponíveis para que o script obtenha um bom tempo de execução. Este script não será exibido no momento pois envolve conceitos além dos vistos, ficando para didáticas futuras sua implementação.

Até agora falamos e utilizamos alguns **imports**, mas o que são eles?

IMPORTS

O Python possui vários arquivos e estes arquivos são chamados de **módulos**. Cada arquivo desse pode possuir **classes,** variáveis, listas etc.… ou seja, são arquivos que possuem códigos!

Mas por que que isso é importante?

Estes arquivos servem de apoio para não ser preciso implementar uma funcionalidade manualmente sempre que for preciso, pois eles já possuem o código pronto e basta “importar” para o programa atual que está utilizando.

Alguns módulos como o socket que utilizamos anteriormente são arquivos que o Python trás em sua instalação. Para identificarmos o caminho de algum desses arquivos podemos contar com o recurso da função print:

```
>>> import socket
>>> print (socket.__file__)
/usr/lib/python3.7/socket.py
```

Identificamos que o arquivo **socket.py** está localizado (no meu caso) em: **/usr/lib/python3.7/**

Listando este diretório pode ser identificado muitos outros arquivos:

```
root@desecsecurity:~/python# ls /usr/lib/python3.7/
Display all 204 possibilities? (y or n)
```

Além do **socket.py** existem outros 203 arquivos (módulos) que podem ser utilizados para importação.

Entendendo então que os módulos do python que utilizamos para **imports** são apenas arquivos que nos auxiliam a não ser preciso ficar “criando” várias vezes o mesmo código, podemos então criar um módulo com uma função única nossa e utilizá-lo?

Sim!!

Mas antes de tudo precisamos entender como funciona, e como criar uma função.

O que é uma Função?
Algumas vezes utilizamos a mesma sequência de comandos diversas vezes, porém fica ruim ter que reescrevê-la sempre que necessário. Para isso podemos criar uma função, onde armazenaremos esta

sequência de comandos e os chamaremos por um “nome” (pelo nome da função) sempre que precisarmos utiliza-los.

A estrutura de uma função é a seguinte:

def nome_da_funcao():

bloco_de_instrucao

Dessa forma, em qualquer parte do código podemos simplesmente chamar por “nome_da_funcao()” que será executado as instruções dentro dela.

Sabendo disso vamos inserir o nosso Port Scanner em uma função:

```
#!/bin/python3
import socket

print("Learning Python - Desec Security")


def port_scan():
    ip = '37.59.174.225'

    for porta in range(1,100):
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        socket.setdefaulttimeout(0.5)
        if (sock.connect_ex((ip,porta)) == 0):
            print("[+] Porta TCP Aberta:",porta)
            sock.close()

port_scan()
```

Legal! Agora o nosso Port Scanner virou uma função!

Podemos ainda passar **argumentos** para a função. Os argumentos são valores que serão trabalhados conforme o código que há dentro da função.

Em nosso Port Scanner sabemos que precisamos de um IP e uma Porta, então vamos criar um meio para que a função seja chamada e requisite um “IP” como parâmetro.

Para isso precisamos modificar algumas coisas, como criar uma variável para armazenar o IP:

```
ip = input("Insira o IP do alvo: ")
```

Agora que temos a variável “ip” precisamos informar para a função que, para ela funcionar, ela precisa de um argumento: o IP!

```
port_scan(ip)
```

Agora ao invocar a função estamos passando a variável IP, informando a ela que o IP será utilizado como parâmetro:

E executando, dá erro:

```
root@desecsecurity:~/python# ./portscan.py
Learning Python - Desec Security

Insira o IP do alvo: 37.59.174.225
Traceback (most recent call last):
  File "./portscan.py", line 16, in <module>
    port_scan(ip)
TypeError: port_scan() takes 0 positional arguments but 1 was given
```

Isso ocorre por que precisamos programar a função para ela receber um valor como argumento e trabalhar com ele... Para isso, na criação da função precisamos criar uma “variável de referência” do parâmetro que estamos passando:

```
 1 #!/bin/python3
 2 import socket
 3
 4 print("Learning Python - Desec Security\n")
 5
 6 ip = input("Insira o IP do alvo: ")
 7
 8 def port_scan(alvo):
 9     for porta in range(1,100):
10         sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
11         socket.setdefaulttimeout(0.5)
12         if (sock.connect_ex((alvo,porta)) == 0):
13             print("[+] Porta TCP Aberta:",porta)
14             sock.close()
15
16 port_scan(ip)
```

Na linha 8, informamos que a função receberá algum valor**:**

```
 8 def port_scan(alvo):
```

Na linha 16 onde invocamos a função, estamos passando a variável IP como parâmetro:

```
16 port_scan(ip)
```

Estamos informando então que a função precisa de um argumento, e estamos passando o IP. Porém na função não iremos operar diretamente com a variável **ip**. O nome do parâmetro (neste caso “**alvo**”) irá receber o valor de **ip**.

Foi então modificado a **tupla**, informando que o IP do alvo é o valor do “**alvo**”:

```
12         if (sock.connect_ex((alvo,porta)) == 0):
```

Talvez tenha ficado confuso, mas para vermos de uma forma visual vamos utilizar o print para exibir o que ocorreu:

```
 1 #!/usr/bin/python3
 2
 3 ip = input("IP: ")
 4
 5 print("Variavel ip:",ip)
 6
 7 def port_scan(alvo):
 8     print("Valor da variavel 'alvo':",alvo)
 9
10 port_scan(ip)
```

Ao executarmos, vemos que o “alvo” assumiu o valor de “ip”:

```
root@desecsecurity:~/python# python3 exemplo.py
IP: 37.59.174.225
Variavel ip: 37.59.174.225
Valor da variavel 'alvo': 37.59.174.225
```

E assim é como declara e utiliza uma função!

Criando Módulos no Python

Criamos uma função para o Port Scanner, agora toda vez que formos utilizá-lo devemos invocar a função que contém seu código.

Além de um IP o nosso Port Scanner precisa de uma lista de portas. Passar as portas através de um range funciona, porém podemos criar uma lista personalizada para utilizar em nosso Scan. Para isso vamos criar uma lista com essas portas!

Vamos criar um novo arquivo, com uma lista de portas no mesmo diretório que está o script de nosso Port Scanner:

```
root@desecsecurity:~/python# ls
informacoes.py  portscan.py
root@desecsecurity:~/python# cat informacoes.py
portas_top10 = [21,22,23,25,80,110,139,443,445,3389]
```

Como vimos, um módulo é um arquivo com instruções que pode ser importado! Sabendo disso, podemos importar o arquivo **informacoes.py** para o nosso port scanner:

```
1 #!/bin/python3
2 import socket
3 import informacoes
```

Após importado o arquivo **informações**, podemos utilizar os recursos dentro dele. No momento temos uma wordlist de portas.

Para utiliza-la, basta invocar utilizando como sintaxe:

nome_do_arquivo.recurso

Neste exemplo ficaria: informacoes.port_top10

Vamos inserir em um print no início do arquivo apenas para ver sendo invocado:

```
 1 #!/bin/python3
 2 import socket
 3 import informacoes
 4
 5 print("\nLearning Python - Desec Security\n")
 6
 7 print("Lista de portas:",informacoes.portas_top10)
 8
 9 ip = input("Insira o IP do alvo: ")
10
11 def port_scan(alvo):
12     for porta in range(1,100):
13         sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
14         socket.setdefaulttimeout(0.5)
15         if (sock.connect_ex((alvo,porta)) == 0):
16             print("[+] Porta TCP Aberta:",porta)
17             sock.close()
18
19 port_scan(ip)
```

Executando:

```
root@desecsecurity:~/python# ./portscan.py

Learning Python - Desec Security

Lista de portas: [21, 22, 23, 25, 80, 110, 139, 443, 445, 3389]
Insira o IP do alvo:
```

Vemos que funcionou corretamente, e agora podemos utilizar os recursos do nosso novo módulo importado!

Um outro detalhe é que quando é criado módulos locais uma pasta chamada **__pycache__** também é criada (ou em alguns casos, é preciso criar manualmente). Ela armazenará os dados temporários de execução do script.

Sabendo agora que podemos utilizar recursos dos módulos importados! Vamos então criar melhorias no script para ele utilizar esta wordlist de portas.

Vamos pensar no cenário das portas... O que precisamos fazer?

1 – Criar Listas com várias portas a serem testadas (as top 10 por exemplo);

2 – Enviar uma mensagem intuitiva para o usuário para ele escolher uma das listas;

3 – Criar uma tomada de decisão eficiente no programa, para ele utilizar a lista escolhida pelo usuário.

Partindo dessa ideia podemos criar primeiro 3 listas distintas, contendo várias portas cada uma:

```
portas_top10 = [21,22,23,25,80,110,139,443,445,3389]
portas_top20 = [135,139,143,443,445,993,995,1723,3306,3389,5900,8080]
portas_top50 = [3389,5060,5666,5900,6001,8000,8008,8080,8443,8888,10000,32768,49152,49154]
```

Perfeito, temos agora três listas. O nome delas é bem intuitivo (no caso de exemplo, fictício para ficar melhor a foto, no script disponibilizaremos na integra).

Agora é preciso informar ao usuário que ele possui três opções para a escolha, sendo a lista “top10”, “top20” e “top50”. Para isso podemos enviar algumas mensagens intuitivas na tela:

```
print("Selecione uma Lista de Portas: ")
print("1 - Lista top 10")
print("2 - Lista top 20")
print("3 - Lista top 50")
escolha = input("\n>>> ")
```

Nas mensagens acima, o usuário verá as opções na tela e a sua escolha será armazenada na variável “escolha”.

Bem, agora falta apenas a tomada de decisão correto? Precisamos criar uma condição para que quando o usuário digitar algum dos valores, a lista correspondente seja utilizada. Podemos utilizar o **if** para realizar esta ação:

```
lista_escolhida = []

if (escolha == '1'):
    lista_escolhida = portas_top10
    return lista_escolhida
elif (escolha == '2'):
    lista_escolhida = portas_top20
    return lista_escolhida
else:
    lista_escolhida = portas_top50
    return lista_escolhida
```

Para a nossa tomada de decisão funcionar primeiro precisamos declarar uma lista vazia, pois não vamos simplesmente enviar a lista original para o port scanner. Em seguida entra o nosso **if.** Caso o usuário escolha a opção 1 por exemplo, nossa "lista_escolhida" irá receber os elementos que estão na lista "portas_top10", e em seguida será enviada como resposta da função pela instrução **return.**

> **O return serve para retornar o resultado de uma determinada operação que ocorreu em uma função, assim podemos coletar apenas o valor final após o código ter sido executado. Neste caso pegaremos apenas a instrução final que nos importa: A lista escolhida.**

Vamos organizar todo o código da seleção de portas dentro de uma função, dessa forma podemos invocar esta função para ser tomada a decisão.

O arquivo **configuracoes.py** ficou então dessa forma:

```
1  def escolher_listaPortas():
2      portas_top10 = [21,22,23,25,80,110,139,443,445,3389]
3      portas_top20 = [135,139,143,443,445,993,995,1723,3306,3389,5900,8080]
4      portas_top50 = [3389,5060,5666,5900,6001,8000,8008,8080,8443,8888,10000,32768,49152,49154]
5
6      print("Selecione uma Lista de Portas: ")
7      print("1 - Lista top 10")
8      print("2 - Lista top 20")
9      print("3 - Lista top 50")
10     escolha = input("\n>>> ")
11
12     lista_escolhida = []
13
14     if (escolha == '1'):
15         lista_escolhida = portas_top10
16         return lista_escolhida
17     elif (escolha == '2'):
18         lista_escolhida = portas_top20
19         return lista_escolhida
20     else:
21         lista_escolhida = portas_top50
22         return lista_escolhida
```

Dessa forma temos implementado:

Listas de portas;

"Menu" de seleção da lista que será utilizada;

Tomada de decisão, enviando como retorno a lista escolhida.

E agora vamos aplicar a nova lista de tomada de decisões em nosso arquivo **portscan.py:**

```
1 #!/bin/python3
2 import socket
3 import informacoes
4
5 print("\nLearning Python - Desec Security\n")
6
7 def port_scan():
8     ip = input("Insira o IP do alvo: ")
9     lista_portas = informacoes.escolher_listaPortas()
10    print("Iniciando os scans no IP: ",ip)
11    for porta in lista_portas:
12        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
13        socket.setdefaulttimeout(0.5)
14
15        if (sock.connect_ex((ip,porta)) == 0):
16            print("[+] Porta TCP Aberta:",porta)
17            sock.close()
18
19 port_scan()
```

Acima vemos como ficou o código do Port scanner. Como o arquivo **informacoes.py** virou um "modulo", precisamos importa-lo para utilizar seus recursos. Isso ocorreu na linhs 3 através do "**import informacoes**":

```
3 import informacoes
```

Aqui é um ponto interessante, pois quando realizamos um import dessa forma estamos importando **TODOS** os recursos do módulo. **TODOS.** Em nosso caso há apenas uma função específica, porém imagina se houvessem 10? Ou várias variáveis (sim, pois podemos usar as variáveis também)... Enfim, poderia dar problema no arquivo. Sabendo disso podemos deixar explicito que queremos importar **APENAS** a função que nos interessa (ou a parte do código que queremos).

A sintaxe seria dessa forma:

from informacoes **import** escolher_listaPortas()

Assim apenas esta função funcionaria em nossa referência, e caso houvesse mais códigos no arquivo precisaríamos importá-los também.

Prosseguindo, na linha 5 temos a mensagem de boas vindas, e na linha 7 temos a nossa função do Port Scanner.

A tomada de decisão sobre a lista de portas utilizadas ocorreu neste trecho do código:

```
7 def port_scan():
8     ip = input("Insira o IP do alvo: ")
9     lista_portas = informacoes.escolher_listaPortas()
10    print("Iniciando os scans no IP: ",ip)
11    for porta in lista_portas:
```

Na linha 9 estamos invocando a função **escolher_listaPortas()** que está localizada no arquivo **informacoes**.py. Ao invocar esta função, foi criado logo no início da linha uma lista chamada **lista_portas** para coletar os valores que será recebido como return da função de escolha.

Vendo na prática como ficou vamos executar o **portscan.py**:

```
root@desecsecurity:~/python# ./portscan.py

Learning Python - Desec Security

Insira o IP do alvo: 37.59.174.225
Selecione uma Lista de Portas:
1 - Lista top 10
2 - Lista top 20
3 - Lista top 50

>>> 1
Iniciando os scans no IP:  37.59.174.225
[+] Porta TCP Aberta: 21
[+] Porta TCP Aberta: 22
[+] Porta TCP Aberta: 80
```

Ao executa-lo funcionou corretamente! Ele solicitou um IP e exibiu o menu para seleção da lista de portas que seria utilizada. Ao selecionar a “top 10” com a opção 1, o scan ocorreu rapidamente.

Podemos deixar um pouco mais intuitivo, fazendo ele nos informar quais portas serão utilizadas para testes:

```
10      print("Iniciando os scans no IP: ",ip)
11      print("A lista escolhida possui as seguintes portas:\n",lista_portas,"\n")
```

Agora temos uma saída com maiores informações:

```
root@desecsecurity:~/python# ./portscan.py

Learning Python - Desec Security

Insira o IP do alvo: 37.59.174.225
Selecione uma Lista de Portas:
1 - Lista top 10
2 - Lista top 20
3 - Lista top 50

>>> 1
Iniciando os scans no IP:  37.59.174.225
A lista escolhida possui as seguintes portas:
 [21, 22, 23, 25, 80, 110, 139, 443, 445, 3389]

[+] Porta TCP Aberta: 21
[+] Porta TCP Aberta: 22
[+] Porta TCP Aberta: 80
```

Neste momento você tem um Port Scanner funcional, e pode realizar manipulações através de imports locais! Vai agora de sua criatividade na manipulação dos dados para uma saída mais organizada. Também poderá verificar modos diferentes de requisições TCP para a validação do estado de uma porta!

Com isso terminamos o nosso Port Scanner, espero que tenham aprendido (ou relembrado) coisas legais sobre o Python e algumas das coisas simples que podemos fazer de forma manual.

Segue as fotos de como ficaram os nossos dois arquivos.

Arquivos na pasta do projeto:

```
root@desecsecurity:~/python# ls
informacoes.py  portscan.py  __pycache__
```

portscan.py

```python
#!/bin/python3
import socket
import informacoes

print("\nLearning Python - Desec Security\n")

def port_scan():
    ip = input("Insira o IP do alvo: ")
    lista_portas = informacoes.escolher_listaPortas()
    print("Iniciando os scans no IP: ",ip)
    print("A lista escolhida possui as seguintes portas:\n",lista_portas,"\n")
    for porta in lista_portas:
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        socket.setdefaulttimeout(0.5)

        if (sock.connect_ex((ip,porta)) == 0):
            print("[+] Porta TCP Aberta:",porta)
            sock.close()

port_scan()
```

Informacoes.py

```python
def escolher_listaPortas():
    portas_top10 = [21,22,23,25,80,110,139,443,445,3389]
    portas_top20 = [135,139,143,443,445,993,995,1723,3306,3389,5900,8080]
    portas_top50 = [3389,5060,5666,5900,6001,8000,8008,8080,8443,8888,10000,32768,49152,49154]

    print("Selecione uma Lista de Portas: ")
    print("1 - Lista top 10")
    print("2 - Lista top 20")
    print("3 - Lista top 50")
    escolha = input("\n>>> ")

    lista_escolhida = []

    if (escolha == '1'):
        lista_escolhida = portas_top10
        return lista_escolhida
    elif (escolha == '2'):
        lista_escolhida = portas_top20
        return lista_escolhida
    else:
        lista_escolhida = portas_top50
        return lista_escolhida
```