Estratégia Concursos

Aula 04

Banco do Brasil (Escriturário - Agente de Tecnologia) Passo Estratégico de Tecnologia de Informação - 2023 (Pós-Edital)

Autor:
Thiago Rodrigues Cavalcanti

31 de Janeiro de 2023

Python 3.9.X aplicada para IA/ML e Analytics (bibliotecas Pandas, NumPy, SciPy, Matplotlib e Scikit-learn). - Parte 1

Sumário

ANÁLISE ESTATÍSTICA

Inicialmente, convém destacar os percentuais de incidência de todos os assuntos previstos no nosso curso – quanto maior o percentual de cobrança de um dado assunto, maior sua importância:

Assunto	Quantidade	Grau de incidência em concursos similares CESGRANRIO
Modelagem conceitual de dados (a abordagem entidade-relacionamento); Modelo relacional de dados (conceitos básicos, normalização);	55	23,40%
Linguagem SQL2008;	49	20,85%
5. Estrutura de dados e algoritmos: Busca sequencial e busca binária sobre arrays; Ordenação (métodos da bolha, ordenação por seleção, ordenação por inserção), lista encadeada, pilha, fila e noções sobre árvore binária.	42	17,87%
6. Ferramentas e Linguagens de Programação para manipulação de dados: Ansible; Java (SE 11 e EE 8); TypeScript 4.0;	34	14,47%
Data Warehouse (modelagem conceitual para data warehouses, dados multidimensionais);	31	13,19%
Python 3.9.X aplicada para IA/ML e Analytics (bibliotecas Pandas, NumPy, SciPy, Matplotlib e Scikit-learn).	11	4,68%
2. Banco de Dados: Banco de dados NoSQL (conceitos básicos, bancos orientados a grafos, colunas, chave/valor e documentos);	4	1,70%
4. Desenvolvimento Mobile: linguagens/frameworks: Java/Kotlin e Swift. React Native 0.59; Sistemas Android api 30 e iOS xCode 10.	4	1,70%
3. Big data: Fundamentos; Técnicas de preparação e apresentação de dados.	3	1,28%
1. Aprendizagem de máquina: Fundamentos básicos; Noções de algoritmos de aprendizado supervisionados e não supervisionados	1	0,43%
Postgre-SQL;	1	0,43%
Noções de processamento de linguagem natural.	0	0,00%
Conceitos de banco de dados e sistemas gerenciadores de bancos de dados (SGBD); MongoDB;	0	0,00%

ROTEIRO DE REVISÃO E PONTOS DO ASSUNTO QUE MERECEM DESTAQUE

A ideia desta seção é apresentar um roteiro para que você realize uma revisão completa do assunto e, ao mesmo tempo, destacar aspectos do conteúdo que merecem atenção.

Dicionário

Faremos uma lista de termos que são relevantes ao entendimento do assunto desta aula! Se durante sua leitura texto, você tenha alguma dúvida sobre conceitos básicos, esta parte da aula pode ajudar a esclarecer.

Python

'**lambda**' é uma **palavra-chave** que permite **criar funções lambda**. Estas são funções anônimas usadas para **realizar cálculos simples**. **Exemplo,**

>>> x = lambda a, b: a + b

>>> x (5, 6)

Agora, **x** é uma **função** que recebe **dois argumentos** e **retorna sua soma**.

Variável é uma maneira de armazenar valores na memória do computador usando nomes específicos que você define.

Tipos de dados

Inteiro (**int**) = Número inteiro

Float (**float**) = número decimal

String (**str**) = Texto

Boolean (**bool**) = True/False

List (**list**) = Um "contêiner" que pode armazenar qualquer tipo de valor. Você pode criar uma lista com colchetes, por exemplo: [1, 2, 3, 'a', 'b', 'c']

Tupla (**tuple**) = Um "contêiner" semelhante à lista com a diferença de que você **não pode atualizar** os valores em uma tupla. Você pode criar uma tupla com **parênteses:** (1, 2, 3, 'a', 'b', 'c')

Número do índice (index number) é a localização do valor específico armazenado em listas ou tuplas do Python. **O primeiro valor de índice da lista é sempre 0.**

Script é um documento dedicado para escrever código Python que você pode executar. Arquivos de script Python devem sempre ter a extensão de arquivo `.py `.

Comprimento da lista - O comprimento da lista refere-se ao **tamanho** de uma **lista** , ou seja, o **número de elementos na lista** .

Loader - Um loader é um **objeto** que **carrega** um **módulo** que está sendo **importado**.

Loop - Um loop é usado para **iterar** sobre uma **sequência** (como **list**, **tuple** , **string**, **dictionary**, **set**). Python oferece dois loops: **for** e **while** .

Loops aninhados referem-se a um **loop dentro de outro loop** . Para **cada passagem** do **loop externo**,

Métodos - Métodos Python são **funções associadas** a um **objeto**. Os métodos de um **objeto operam** nos **dados** desse **objeto**. Chamamos **um método** em um **objeto** usando o **operador ponto**. Por exemplo, se **obj** é um **objeto** e **med()** é seu **método**, nós o chamamos de **obj.med()** .

Módulo - *module* é um **arquivo que consiste** em **código Python**. Este módulo pode ser **importado** e o **código dentro** dele pode ser **reutilizado** em outro **programa Python**.

Mutável - Um tipo de dado mutável é aquele que pode **mudar** seu **valor** durante a **execução do programa** .

o **loop interno é executado** completamente antes **de passar** para a **próxima passagem** do **loop externo** .

Objeto - Um objeto é uma **coleção de dados** e **métodos**. Objetos modelam **entidades do mundo real** em **programação orientada a objetos**.

Orientado a objetos - A programação orientada a objetos é um **paradigma de programação** baseado em **objetos**. A ênfase na programação orientada a objetos está nos **dados** e não **nos procedimentos**.

Parâmetro - Um parâmetro é uma **variável** especificada em uma **definição de função** entre **parênteses**. Parâmetro denota os **valores de argumento que** uma **função pode aceitar**.

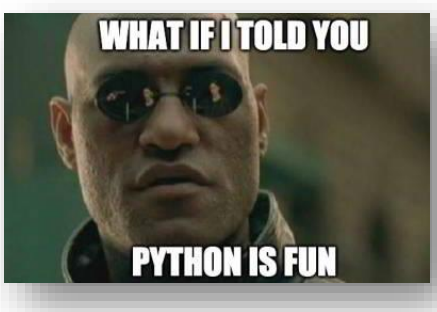
Slice - Refere-se ao acesso a uma **parte específica** de uma **sequência**, como **listas**, **tuplas** e **strings**. O fatiamento pode ser usado para **modificar** e **manipular partes** de **sequências mutáveis,** como **listas**.

Virtual machine - A máquina virtual Python é responsável por **executar** o **bytecode gerado** a partir do **código fonte Python** .

Para revisar e ficar bem-preparado no assunto, você precisa, basicamente, seguir os passos a seguir:

LINGUAGEM PYTHON

Introdução



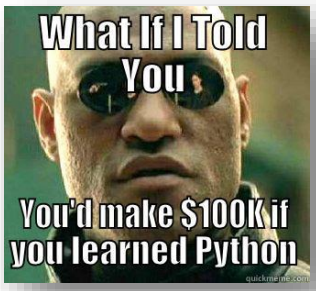


Figura 1 - Python é extremamente divertido e desenvolvedores em Python chegam a ganhar 100 mil dólares por ano.

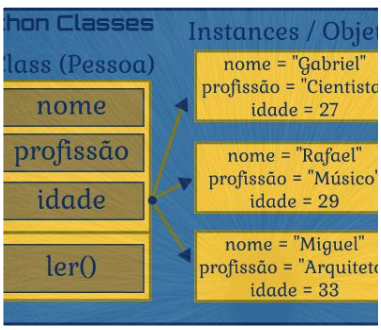
Quando escrevemos código, estamos instruindo um computador sobre as coisas que ele deve fazer. Se você escrever um pedaço de software que permite que as pessoas comprem roupas online, você terá que representar pessoas reais, roupas reais, marcas reais, tamanhos e assim por diante, dentro dos limites de um programa.

Para fazer isso, **você precisará criar e manipular objetos no programa que está sendo escrito**. Uma pessoa pode ser um objeto. Um carro é um objeto. Um par de calças é um objeto.

As duas principais características de qualquer objeto são **propriedades** e **métodos**. Tomemos o exemplo de uma pessoa como objeto. Normalmente, em um programa de computador, você representa as pessoas como clientes ou funcionários. As propriedades que você armazena neles são coisas como um nome, um número de CPF, uma idade, se eles têm carteira de motorista, um e-mail, sexo e assim por diante. Na figura abaixo vemos uma pessoa do mundo real.

Churrasqueiro

Mundo real



Prestador de serviço

Objeto

Em um programa de computador, você armazena todos os dados necessários para usar um objeto para a finalidade que precisa ser atendida. Se você está codificando um site para vender roupas, provavelmente deseja armazenar as alturas e pesos, bem como outras medidas de seus clientes, para que as roupas apropriadas possam ser sugeridas a eles. Assim, as propriedades são características de um objeto. Nós as usamos o tempo todo: *Você poderia me passar essa caneta? - Qual delas? — a preta*. Aqui, usamos a cor (*preta*) de uma caneta para identificá-la (provavelmente ela estava sendo mantida ao lado de canetas de cores diferentes para que a distinção fosse necessária).

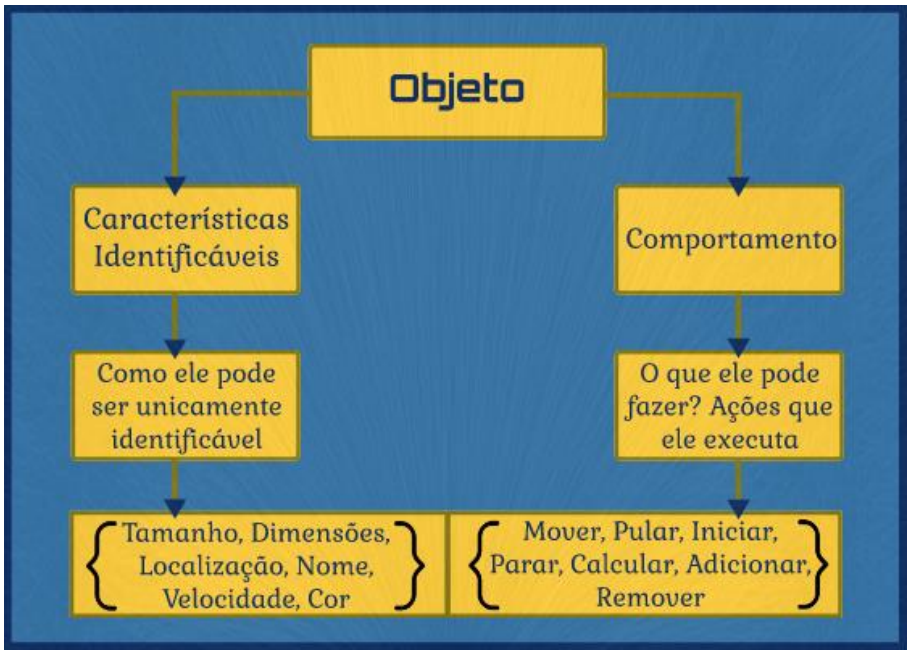


Figura 2 - Definição de objeto

Agora que você já entende as características de um objeto (atributos), vamos tratar do comportamento. Neste sentido, precisamos entender o que são métodos. Métodos são **coisas que um objeto pode fazer**. Como pessoa, tenho métodos como *falar, andar, dormir, acordar, comer, sonhar, escrever, ler* e assim por diante. Todas as coisas que posso fazer podem ser vistas como métodos dos objetos que me representam.

Então, agora que você sabe o que são objetos, que eles expõem métodos que podem ser executados e propriedades que você pode inspecionar, você está pronto para começar a codificar. Codificar, na verdade, é simplesmente gerenciar aqueles objetos que vivem no subconjunto do mundo que estamos reproduzindo em nosso software. Você pode criar, usar, reutilizar e excluir objetos como quiser. Segundo a documentação:

"Os objetos são a abstração de dados do Python. Todos os dados em um programa Python são representados por objetos ou por relações entre objetos."

Veremos mais detalhadamente os objetos Python mais adiante, nesta aula. Por enquanto, tudo o que precisamos saber é que todo objeto em Python tem um **ID** (ou identidade), um **tipo** e um **valor**.

Uma vez criado, o ID de um objeto nunca é alterado. É um identificador exclusivo para ele e é usado nos bastidores pelo Python para recuperar o objeto quando quisermos usá-lo. O tipo também nunca muda. O

tipo indica quais operações são suportadas pelo objeto e os possíveis valores que podem ser atribuídos a ele. Veremos os tipos de dados mais importantes do Python em instantes. O valor pode ser alterado ou não: se puder, o objeto é dito ser **mutável**; caso contrário, diz-se que é **imutável**.

Como, então, podemos usar um objeto? Damos-lhe um nome, claro! Quando você dá um nome a um objeto, então você pode usar o nome para recuperar o objeto e usá-lo. Em um sentido mais genérico, objetos, como números, strings (texto) e coleções, são associados a um nome. Normalmente, dizemos que esse nome é o **nome de uma variável**. Você pode ver a variável como uma caixa, que pode ser usada para armazenar dados.

Assim, você tem todos os objetos de que precisa; e agora? Bem, precisamos usá-los, certo? Podemos querer enviá-los por uma conexão de rede ou armazená-los em um banco de dados. Talvez os exibir em uma página da web ou gravá-los em um arquivo. Para isso, precisamos reagir a um usuário preenchendo um formulário, ou pressionando um botão, ou abrindo uma página da web e realizando uma pesquisa. Reagimos **executando nosso código**, **avaliando condições** para escolher quais partes executar, quantas vezes e em quais circunstâncias.

Para fazer tudo isso, precisamos de uma linguagem. É para isso que serve o Python. Python é a linguagem que usaremos nesta aula para instruir o computador a fazer algo por nós. Sobre Python:

- Python é uma **linguagem de programação** poderosa e fácil de aprender.

- Possui estruturas de dados de alto nível eficientes e uma abordagem simples, mas eficaz para a **programação orientada a objetos**.

Certo! Mas para que exatamente serve Python?

A sintaxe elegante e a **tipagem dinâmica** do Python, junto com sua natureza **interpretada**, tornam-no uma linguagem ideal para scripts e desenvolvimento rápido de aplicativos em muitas áreas na maioria das plataformas.

O interpretador Python e a extensa biblioteca padrão estão **disponíveis gratuitamente** na forma de código-fonte ou binário para todas as principais plataformas no site do Python, *https://www.python.org/*. O mesmo site também contém distribuições e indicadores para muitos módulos, programas e ferramentas gratuitos de terceiros, e documentação adicional.

O interpretador Python é facilmente estendido com novas funções e tipos de dados implementados em C ou C ++ (ou outras linguagens que podem ser chamadas de C). Python também é adequado como uma linguagem de extensão para aplicativos personalizáveis. Se quisermos podemos organizar os aspetos positivos do Python da seguinte forma:

Portabilidade - O Python é executado em qualquer lugar, e portar um programa do Linux para o Windows ou Mac geralmente é apenas uma questão de correção caminhos e configurações. Python é projetado para

portabilidade e cuida de peculiaridades específicas **do sistema operacional** (**SO**) por trás de interfaces que protegem você da dor de ter que escrever código sob medida para uma plataforma específica.

Coerência - Python é extremamente lógico e coerente.

Produtividade do desenvolvedor - De acordo com Mark Lutz, um programa Python é tipicamente um quinto a um terço do tamanho do código Java ou C++ equivalente. Isso significa que o trabalho é feito mais rápido. Mais rápido significa ser capaz de responder mais rapidamente ao mercado. Menos código não significa apenas menos código para escrever, mas também menos código para ler (e programadores profissionais leem muito mais do que escrevem), mantêm, depuram e "refatoram".

Outro aspecto importante é que o Python é executado sem a necessidade de etapas longas e demoradas de compilação e vinculação, portanto, não há necessidade de esperar para ver os resultados do seu trabalho.

Uma extensa biblioteca - Python tem uma biblioteca padrão incrivelmente extensa. Se isso não fosse suficiente, a comunidade internacional Python mantém um corpo de bibliotecas de terceiros, adaptadas a necessidades específicas, que você pode acessar livremente no **Python Package Index** (**PyPI**). Quando você codifica Python e percebe que um determinado recurso é necessário, na maioria dos casos, existe pelo menos uma biblioteca onde esse recurso já foi implementado.

Qualidade do software - Python é fortemente focado em legibilidade, coerência e qualidade. A uniformidade da linguagem permite alta legibilidade, e isso é crucial hoje em dia, pois a codificação é mais um esforço coletivo do que um esforço individual. Outro aspecto importante do Python é sua **natureza multi-paradigma intrínseca. Você pode usá-lo como uma linguagem de script, mas também pode explorar estilos de programação orientados a objetos, imperativos e funcionais — é extremamente versátil.**

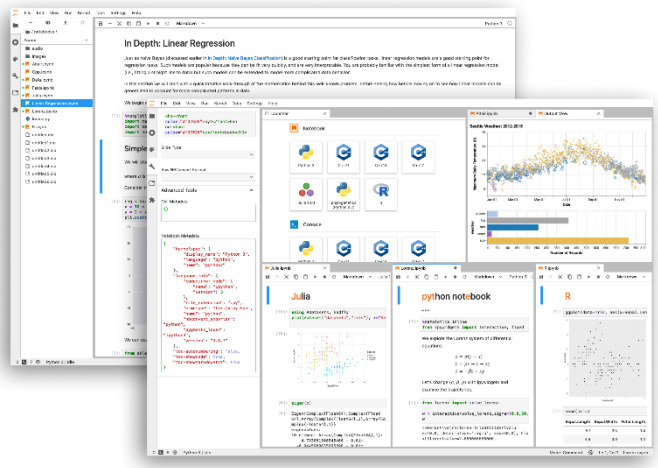
Integração de software - Outro aspecto importante é que o Python pode ser estendido e integrado com muitas outras linguagens, o que significa que mesmo quando uma empresa está usando uma linguagem diferente como principal ferramenta, o Python pode entrar e agir como um agente de interoperabilidade entre aplicativos complexos que precisam conversar entre si de alguma forma. Este é um tópico mais avançado, mas no mundo real, esse recurso é importante.

Satisfação e prazer - Por último, mas não menos importante, existe a diversão! Trabalhar com Python é divertido; podemos codificar por oito horas e sair do escritório felizes e satisfeitos, não afetados pela luta que outros programadores têm que suportar porque usam linguagens que não lhes fornecem a mesma quantidade de estruturas e construções de dados bem projetadas. Python torna a codificação divertida, sem dúvida. E a diversão promove motivação e produtividade.

Esses são os principais aspectos do motivo pelo qual recomendamos o Python para todos. Claro, existem muitos outros recursos técnicos e avançados que poderíamos ter mencionado, mas eles realmente não pertencem a uma seção introdutória. Vejam que essas características, provavelmente, refletem nas escolhas de grandes corporações no sentido de usarem a linguagem para tratar suas demandas por soluções de software. A figura abaixo mostra onde Python vem sendo usada:

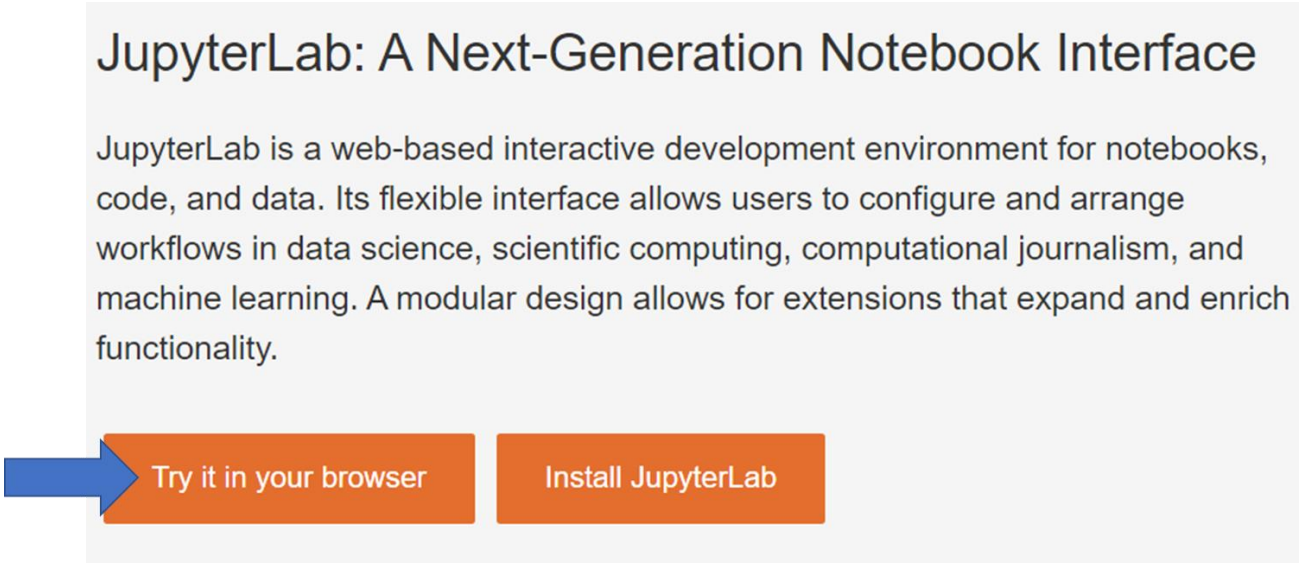
Como Praticar

Depois de ter lido essa rápida introdução você estar se perguntando, por que eu não aprendi essa linguagem tão interessante ainda? Para aprender a linguagem de verdade você vai precisar usá-la, e existem alguns lugares onde você pode começar a escrever seu código. Você pode instalar ou a acessar a versão Web do JupterLab.

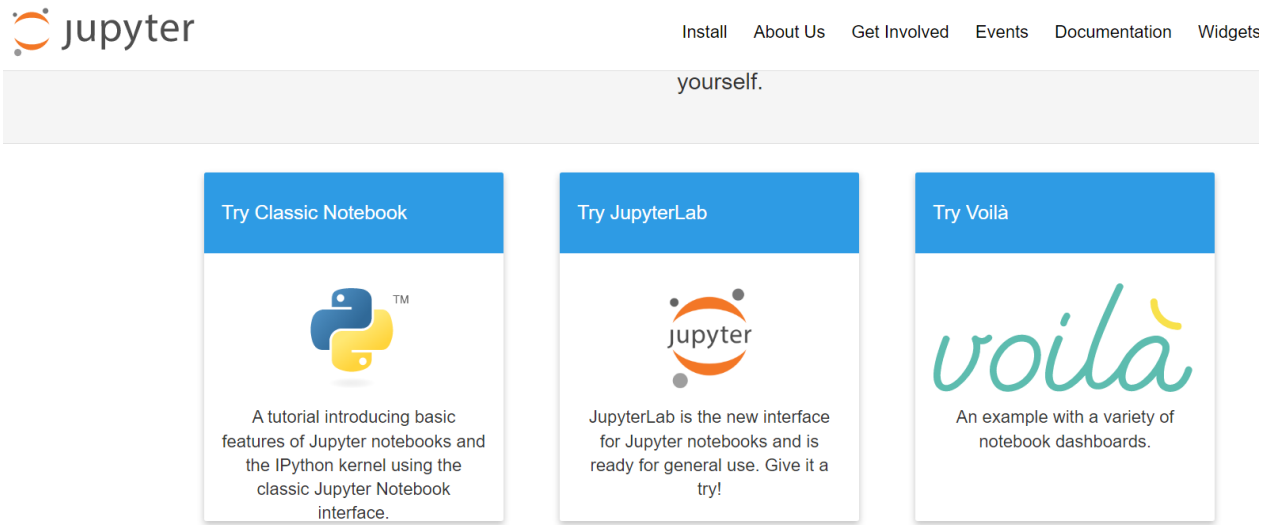


O Project Jupyter desenvolve software de código aberto, padrões abertos e serviços para computação interativa em dezenas de linguagens de programação. JupyterLab é um ambiente de desenvolvimento interativo baseado na web para notebooks, código e dados. Sua interface flexível permite que os usuários configurem e organizem fluxos de trabalho em ciência de dados, computação científica, jornalismo computacional e aprendizado de máquina. Um design modular permite extensões que expandem e enriquecem a funcionalidade.

Para usar o Python 3 você desse entrar no site: *https://jupyter.org/*, e, na página inicial clicar em “try it in yout browser”.[1]

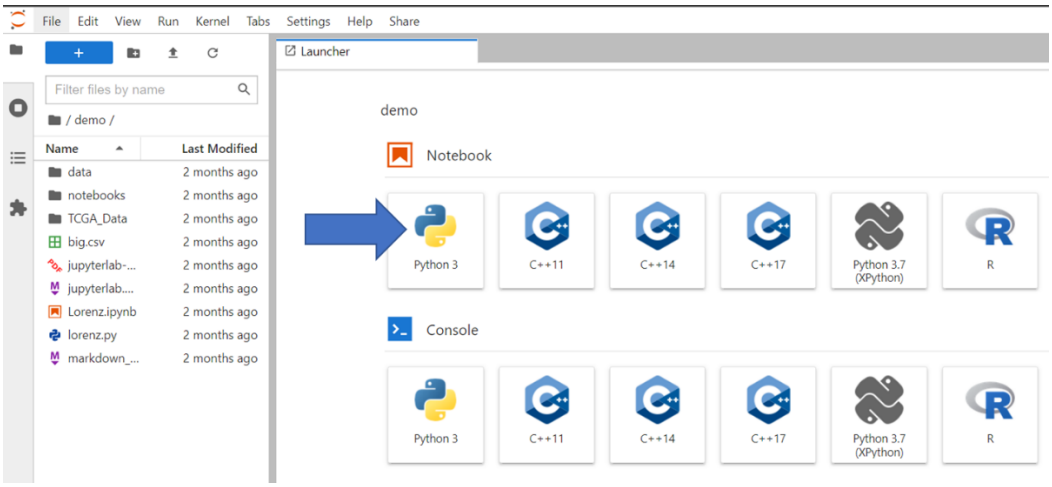


Na tela seguinte você vai clicar em JupterLab:

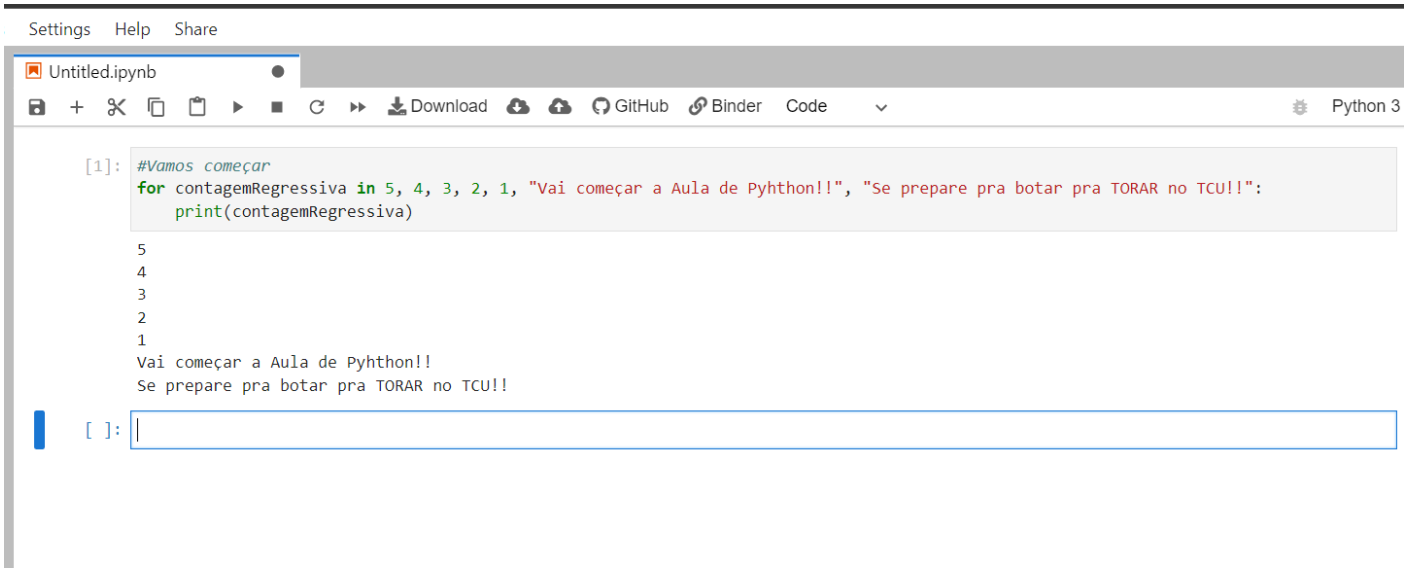


Depois de um processamento rápido do Binder, você verá a tela abaixo, aí basta clicar em Python 3 para começar a codificar.

[1] Outra opção seria baixar uma interface de desenvolvimento (IDE) Python na sua máquina, Spyder e Pycharm são excelentes opções. Basta entrar no site da ferramenta e baixar a versão gratuita da IDE.



A tela de codificação será igual a tela a seguir:



Perceba que escrevemos o código no quadrado cinza, depois apertamos o play para que ele seja interpretado. Logo abaixo, com um fundo branco, aparece a saída do nosso código, que vai imprimir algumas linhas. O código usa uma instrução de repetição (for) para percorrer uma lista de valores e imprimir cada um deles em uma linha diferentes. Perceba, desde já, um aspecto importante da linguagem que é a indentação.

Indentação é uma forma de arrumar o código, fazendo com que algumas linhas fiquem mais à direita que outras, à medida que adicionamos espaços em seu início. Para a maioria das linguagens a indentação não é obrigatória, **mas no caso Python isso é diferente**.

A indentação é uma característica importante no Python, pois **além de promover a legibilidade é essencial para o bom funcionamento do código**. Em outras palavras, se a indentação não estiver adequada o programa pode se comportar de forma inesperada ou até mesmo não compilar.

OK! Agora que já sabemos como colocar em prática o conhecimento, vamos aprender como se comunicar (escrever código) em Python! 😊

Sintaxe

Vamos começar a falar da sintaxe de Python. O que é possível fazer a partir de um conjunto de elementos ou palavras chaves que fazem parte da linguagem. Essas palavras, quando analisadas pelo interpretador, vão instruí-lo a fazer alguma ação de processamento.

Vamos usar o modo interativo do Python para observar o resultado das nossas ações assim que digitamos nosso código. O modo interativo do Python é um de seus recursos mais úteis porque você pode digitar qualquer instrução válida e ver imediatamente o resultado. Isso é útil para depuração e experimentação. Muitas pessoas, usam o Python interativo como sua calculadora de mesa. Por exemplo:

```
>>> 6000 + 4523,50 + 134,25
10657.75
>>> _ + 8192,75
18850.5
>>>
```

Quando você usa o Python interativamente, a variável _ mantém o resultado da última operação. Isso é útil se você quiser usar esse resultado em instruções subsequentes. Esta variável só é definida ao trabalhar interativamente, portanto, não a use em programas salvos.

Você pode sair do interpretador interativo digitando quit() ou o caractere EOF (fim do arquivo). No UNIX, EOF é Ctrl+D; no Windows, é Ctrl+Z.

Se você quiser criar um programa que possa ser executado repetidamente, coloque instruções em **um arquivo de texto**. Por exemplo:

```
# hello.py
print( 'Hello World' )
```

Os arquivos de origem do Python são **arquivos de texto codificados em UTF-8** que normalmente têm um sufixo ***.py***. O caractere ***#*** denota um comentário que se estende até o final da linha. Caracteres internacionais (Unicode) podem ser usados livremente no código-fonte, desde que você use a codificação UTF-8.

Para executar o arquivo *hello.py*, forneça o nome do arquivo ao interpretador da seguinte forma:

```
shell % python3 hello.py
Hello World
```

É comum usar ***#!*** para especificar o interpretador que estamos usando na primeira linha de um programa, assim:

```
#!/usr/bin/env python3
print( 'Hello World' )
```

No UNIX, se você conceder a esse arquivo permissões de execução, poderá executar o programa digitando hello.py em seu shell. No Windows, você pode clicar duas vezes em um arquivo .py ou digitar o nome do programa na linha de comando (CMD) do Windows para iniciá-lo. A linha #!, se fornecida, é usada para escolher a versão do interpretador. A execução de um programa pode ocorrer em uma janela do console que desaparece imediatamente após a conclusão do programa — geralmente antes que você possa ler sua saída. Para depuração, é melhor executar o programa em um ambiente de desenvolvimento Python (como Pycharm).

O interpretador executa as instruções em ordem até chegar ao final do arquivo de entrada. Nesse ponto, o programa termina.

Variáveis

Variáveis são “slots” nos quais podemos armazenar dados. Algumas linguagens, como C#, Visual Basic e outras, exigem que você declare suas variáveis antes de usá-las junto com o tipo de variável que pode ser, por exemplo, Integer ou String. **Python não exige que você faça isso**. O Python usa um esquema chamado “*Duck Typing*”[2]. Isso significa que você não precisa declarar variáveis antes que elas sejam usadas e que você não precisa especificar o que uma variável de tipo é ou será.

Sensibilidade a maiúsculas e minúsculas

Os nomes das variáveis devem começar com uma letra (maiúscula ou minúscula) ou um caractere de sublinhado. Elas não podem começar com um número, espaço ou caractere de sinal.

Tudo em Python faz distinção entre maiúsculas e minúsculas: não apenas as palavras-chave e funções que o Python fornece, mas todas as variáveis que você criar. Por exemplo:

“Print” não é o mesmo que “print”.

Por causa disso, você pode usar a palavra “Print” como uma variável, mas não a palavra “print”. Dito isso, não é uma boa ideia usar nomes de funções ou palavras-chave “re-caseed”.

Nomeação Adequada de Variável

Um nome de variável deve explicar para que a variável é usada. Se você usar nomes de variáveis como “x” ou “q”, isso não informa nada sobre o que a variável faz ou para que serve. Isso é chamado de “autodocumentação”. Há muito tempo, era prática “normal” usar nomes de variáveis de caractere único sempre que possível. Muitas vezes foi para economizar memória. Embora não haja nada que diga que você não pode usar letras únicas ou alguns nomes de variáveis obscuros (e realmente economiza tempo e força

[2] Duck Typing (tipagem do pato) na programação de computadores é uma aplicação do teste do pato — "Se ele anda como um pato e grasna como um pato, então deve ser um pato" — para determinar se um objeto pode ser usado para um propósito específico.

do dedo ao digitar uma grande quantidade de código), isso tornará muito difícil para você ou outra pessoa ler e manter seu código.

Como espaços não são permitidos em nomes de variáveis, muitas pessoas usam o que é chamado de ***Camel Casing*** para tornar os nomes das variáveis mais fáceis de entender. **Camel Casing** (às vezes chamado de ***Pascal Casing***) diz que você coloca a primeira letra de cada palavra em maiúscula e todo o resto em minúscula:

```
EuVouPassarNoConcurso
EstrategiaConcursos
ValorRetornado
```

Uma alternativa é usar sublinhados no lugar de espaços:

```
esse_eh_um_exemplo_de_nome_grande_para_variavel
```

Qualquer uma destas formas é “aceitável”. Depende principalmente do que você deseja usar ou do que seu trabalho usa como padrão. Como afirmei anteriormente, é importante que você use nomes de variáveis que sejam “auto documentados”.

Atribuição

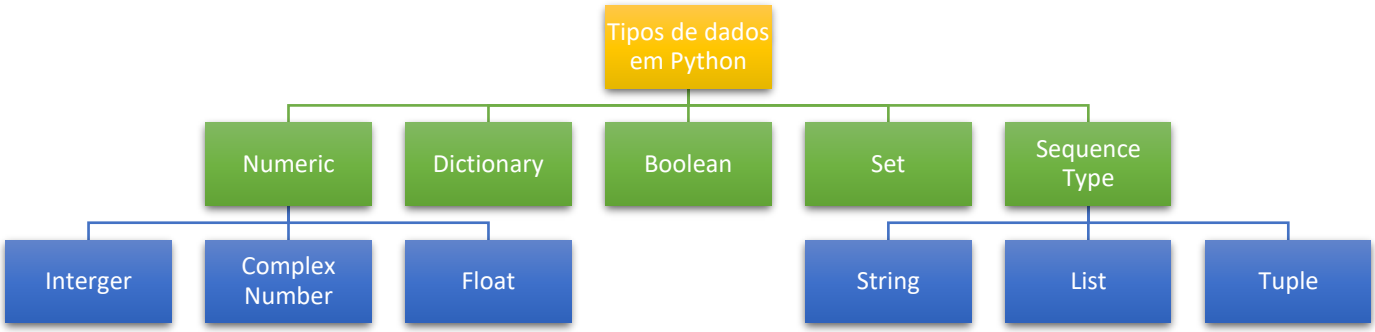
Para atribuir **um valor a uma variável, use o sinal de igual (“=”):**

```
UmInteger = 3
UmFloatValue = 3.14159
UmaString = “Chegou a hora de todos os homens bons…”
UmaLista = [1,2,3,4,5]
```

Você também pode fazer várias atribuições de um único valor em uma linha. No trecho a seguir, a variável 'a' é atribuída ao valor 1. Em seguida, as variáveis 'b' e 'c' são atribuídas para serem iguais à variável 'a:'

```
>>> a = 1
>>> b = c = a
>>> print('a=%d, b=%d, c=%d') % (a,b,c)
a=1, b= 1, c=1
```

Tipos de dados



A figura acima e a tabela abaixo mostram os tipos de dados básicos em Python. Na tabela temos 4 colunas. A primeira apresenta o tipo de dados. A segunda coluna (Tipo) contém o nome Python desse tipo. A terceira coluna (Mutável?) indica se o valor pode ser alterado após a criação. Já a coluna "exemplos" mostra um ou mais exemplos literais desse tipo.

Nome	Tipo	Mutável?	Exemplos
Boolean	`Bool`	não	`True, False`
Integer	`Int`	não	`47, 25000, 25_000`
Floating point	`float`	não	`3.14, 2.7e5`
Complex	`complex`	não	`3j, 5 + 9j`
Text string	`Str`	não	`'estrategia',"festa dos primeiros",'''Thiago Cavalcanti'''`
List	`List`	sim	`[Thiago', 'Rosenval', 'Ricardo Vale']`
Tuple	`tuple`	não	`(2, 4, 8)`
Bytes	`bytes`	não	`b'ab\xff'`
ByteArray	`bytearray`	sim	`bytearray(...)`
Set	`set`	sim	`set([3, 5, 7])`
Frozen set	`frozenset`	não	`frozenset(['Elsa', 'Otto'])`
Dictionary	`dict`	sim	`{'game': 'bingo', 'dog': 'dingo', 'drummer': 'Ringo'}`

Em Python, um objeto é um bloco de dados que contém pelo menos o seguinte:

- Um *tipo* que define o que pode fazer
- Um *id* exclusivo para distingui-lo de outros objetos
- Um *valor* consistente com seu tipo
- Uma *contagem de referência* que rastreia a frequência com que esse objeto é usado

O tipo também determina se o valor de dados contido no objeto pode ser alterado (**mutável**) ou é constante (**imutável**). Pense em um objeto imutável como uma caixa selada, mas com lados claros; você pode ver o valor, mas não pode alterá-lo. Pela mesma analogia, um objeto mutável é como uma caixa com tampa: você pode não apenas ver o valor dentro, mas também alterá-lo.

Python é uma linguagem fortemente tipada e com tipagem dinâmica:

- **Tipagem forte** significa que o tipo de um valor não muda de forma inesperada. Uma *string* contendo apenas dígitos não se torna magicamente um número, como pode acontecer em outras linguagem de programação como *Perl*. **Toda mudança de tipo requer uma conversão explícita**.
- **Tipagem dinâmica** significa que os objetos em tempo de execução (valores) têm um tipo, ao contrário da tipagem estática onde as variáveis têm um tipo.

Agora vamos falar dos principais tipos de dados isoladamente:

Números

Em **Python** existem **três** tipos numéricos, são eles:

- *int*
- *float*
- *complex*

É importante lembrarmos que tudo em **Python** é um **objeto**, sendo assim, tipos de dados são **classes** e **variáveis** são instâncias (objetos) dessas classes. Variáveis do tipo numérico são criadas quando atribuímos valores a elas:

```
a = 27    # int
b = 22.2  # float
c = 3j    # complex
```

Para verificarmos qual classe a variável pertence, podemos utilizar a função **type()**:

```
print(type(a)) # <class 'int'>
print(type(b)) # <class 'float'>
print(type(c)) # <class 'complex'>
```

Int

Inteiros são números **positivos ou negativos** que não apresentam casas decimais, seu tamanho é limitado apenas pela capacidade de memória disponível:

```
a = 3
b = 342907249723902
c = -100
print(type(a)) # <class 'int'>
print(type(b)) # <class 'int'>
print(type(c)) # <class 'int'>
```

Float

Floats ou **"números de ponto flutuante"** são números positivos ou negativos que podem conter uma ou mais casas decimais:

```
a = 23.3
b = 2.0
c = -17.78
print(type(a)) # <class 'float'>
print(type(b)) # <class 'float'>
print(type(c)) # <class 'float'>
```

Podemos acrescentar o caractere **e** seguido por um número inteiro positivo ou negativo para especificar **a notação científica**.

```
e = 35e4
print(type(e)) # <class 'float'>
print(e) # 350000.0
E = 3.8e-2
print(type(E)) # <class 'float'>
print(E) # 0.038
```

O valor máximo que um **float** pode ter é aproximadamente 1.8×10^308. Qualquer número maior que este é indicado pelo valor **inf** (infito), observe:

```
print(1.79e308) # 1.79e+308
print(1.8e308) # inf
```

Entretanto, o valor mínimo positivo que um **float** pode ter é aproximadamente 5.0×10^-324. Qualquer número menor que este é considerado **0** (zero).

```
print(5e-324) # 5e-324
print(5e-325) # 0.0
```

Complex

Números complexos são escritos com j representando a parte imaginária. Eles podem ser escritos complex(3,4) ou 3 + 4j. Um número complexo Python é armazenado internamente usando coordenadas cartesianas ou retangulares. Vejamos alguns exemplos:

```
a = 2+4j
b = -3j
c = complex(3,4)
print(type(a)) # <class 'complex'>
print(type(b)) # <class 'complex'>
print(type(c)) # <class 'complex'>
```

Por fim, podemos obter as partes **Real** e **Imaginária** de um número complexo usando os seguintes atributos do objeto:

```
print(c.real) # 3.0
print(c.imag) # 4.0
```

Operadores numéricos

A linguagem Python nos permite criar expressões matemáticas com imensa facilidade. No dia a dia, nem todos os programas precisarão de expressões matemáticas complexas, mas é importante saber como usar os operadores aritméticos do Python.

Vejamos alguns exemplos de expressões matemáticas em Python.

```
# Soma e subtração
print(10 + 2 - 7)
# Potência: 2 ^ 4
print(2 ** 4)
# Divisão de ponto flutuante
print(20 / 6)
# Divisão inteira, sem considerar o resto
print(20 // 6)
```

A tabela abaixo resume o funcionamento dos operadores aritméticos em Python:

Operador	Descrição	Exemplo
+	soma dois valores	`5 + 2` resulta em `7`
-	subtrai dois valores	`5 - 2` resulta em `3`
*	multiplica dois valores	`5 * 2` resulta em `10`
/	divide dois valores (sem arredondar)	`5 / 2` resulta em `2.5`
//	divide dois valores (arredondando para baixo)	`5 // 2` resulta em `2`
%	resto da divisão	`5 % 2` resulta em `1`
**	exponenciação	`5 ** 2` resulta em `25`

Strings

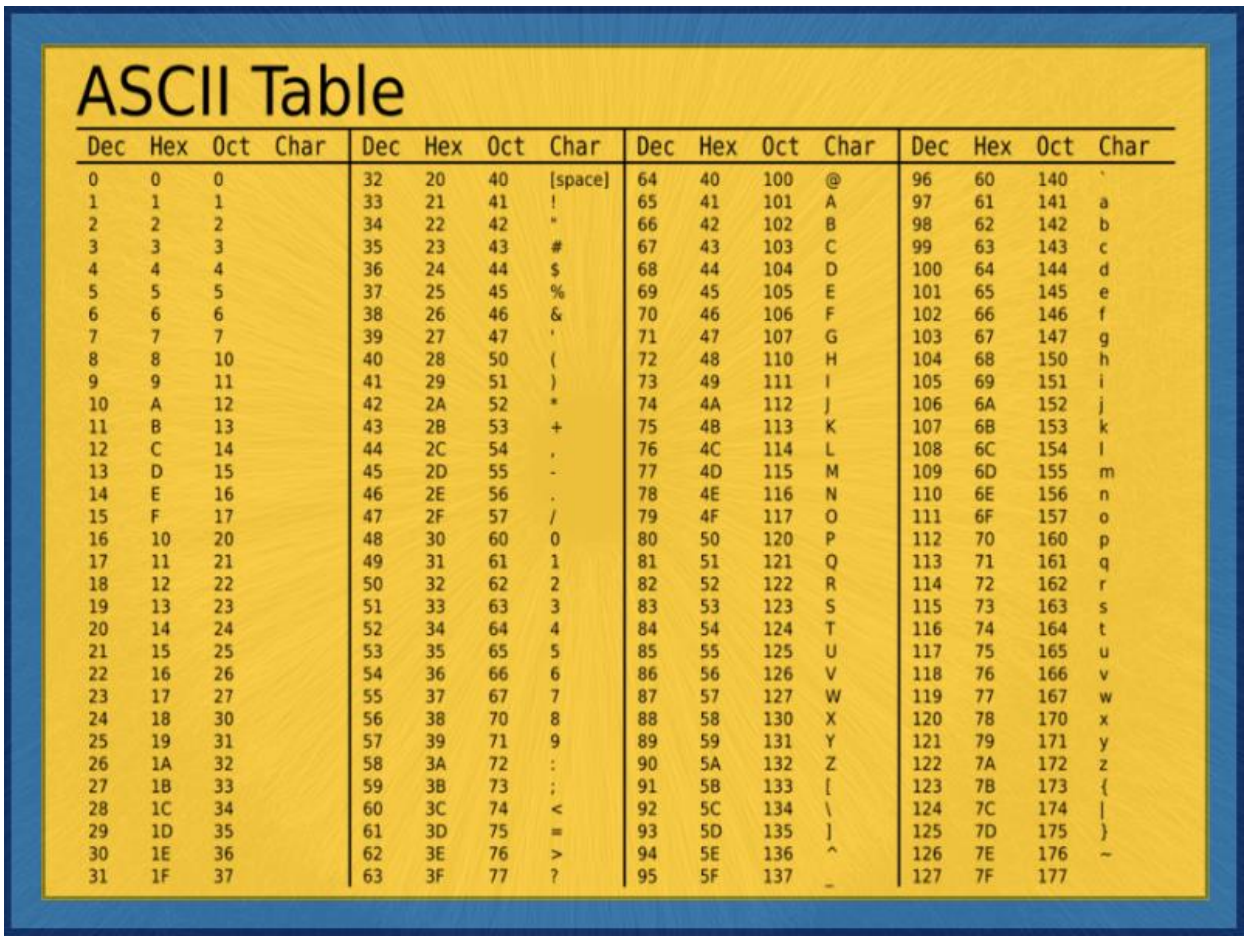
Introdução

Uma **String** é tradicionalmente um tipo de dados que representa uma sequência de caracteres. Um caractere é **simplesmente um símbolo**. Por exemplo, o idioma inglês possui 26 caracteres. Caracteres podem ser **letras**, **dígitos**, **símbolos** ($, !, #, @), etc.

Os computadores originalmente não lidam com caracteres, eles lidam fundamentalmente com números (binários). Mesmo que vejamos caracteres em nossa tela, internamente eles são armazenados e manipulados como uma combinação de **0**'s e **1**'s.

A conversão de caracteres em números é chamada de codificação e o processo reverso é a decodificação. ASCII e Unicode são algumas das codificações populares usadas. Em Python, uma string é uma **sequência de caracteres Unicode**. O Unicode foi introduzido para incluir todos os caracteres em todos os idiomas e trazer uniformidade na codificação, coisa que não era possível utilizando ASCII.

A figura a seguir nos mostra a Tabela ASCII:



ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

Com a função **ord()** podemos obter o valor ASCII para um determinado caractere:

```
ord('A') # 65
ord('X') # 88
ord('<') # 60
```

Contanto que permaneçamos no domínio dos caracteres comuns, haverá pouca diferença prática entre ASCII e Unicode. Mas a função **ord()** retornará valores numéricos para caracteres Unicode também:

```
ord('∑') # 8721
ord('火') # 28779
```

Já o método **chr()** faz o inverso de **ord()**. Fornecido um valor numérico, ele retorna uma string representando este valor.

```
chr(65) # 'A'
chr(88) # 'X'
chr(60) # '<'
chr(28779) # '火'
```

Existem diversos problemas no mundo real que envolvem as **strings**, podemos citar por exemplo:

- Criptografia e Descriptografia
- Análises de DNA
- Tradução de linguagens

Fundamentos

Além dos números, **Python** é capaz de manipular **strings** com maestria, as strings podem ser expressas de diversas maneiras, podemos encapsular elas com aspas simples **('...')** ou também aspas duplas **("...")** ou até mesmo usando 3 aspas **("""...""")** para delimitarmos texto em múltiplas linhas. Por exemplo:

```
print('String é um elemento importante da programação')
print("Também podemos representá-las com aspas duplas")
print("""Podemos, até mesmo, representá-las com três aspas duplas, para
textos grandes""")
```

Como podemos ver, **strings** podem ser impressas utilizando a função **print().** Caso você queira usar palavras entre aspas na sua **string**, você deve usar uma \ **(barra invertida)** para dar um *"escape"* nas aspas, caso contrário ocorrerá um erro. O termos escape é relativamente comum em computação quando queremos desconsiderar as funções tradicionais de caracteres específicos. No nosso caso, não queremos que as aspas sinalizem o final do texto, então colocamos uma barra invertida antes de cada símbolo.

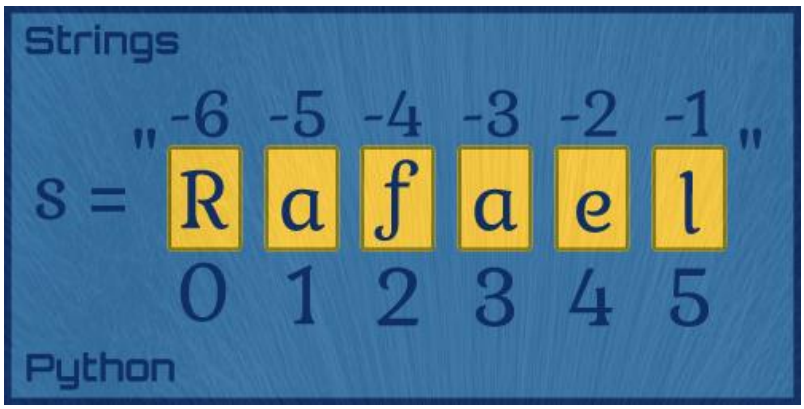
```
print("Podemos usar aspas dentro das \"strings\"")
print('Podemos usar aspas dentro das \'strings\'')
```

Outra solução seria inverter as aspas. Por exemplo, se você usar aspas simples no início da string, pode usar aspas duplas sem problemas no meio do seu texto.

```
print("Dessa forma não há 'problema'")
print('Dessa forma não há "problema"')
```

Índices de Strings (IMPORTANTE!!!)

Assim como em diversas linguagens de programação, **strings** em **Python** são consideradas **arrays de bytes** que representam caracteres unicode, entretanto **Python** não tem um tipo de dados caractere. Caracteres únicos são simplesmente **strings** de tamanho 1. Para acessarmos elementos individuais das **strings**, utilizamos colchetes **[]**. Veja o exemplo abaixo:




```
s = "Rafael"
print(s[0]) # Imprime a primeira letra do nome
print(s[-6]) # Imprime a primeira letra do nome
print(s[5]) # Imprime a última letra do nome
print(s[-1]) # Imprime a última letra do nome
```

Observe que foi utilizado a notação s[0], isso porque o primeiro caractere começa na posição 0, lembre-se que isso é muito comum nas linguagens de programação e estruturas de dados. Cada caractere na **string** é associado com um **índice** numérico, que é um inteiro representando a localização do caractere em uma determinada **string**.

Além disso, podemos formar **substrings** a partir de uma **string**.

```
nome = "John Von Neumann"
Neste exemplo, selecionamos o caracter da posição 5 até 11 (não
incluindo 12):
print(nome[5:12]) # Von Neu
Podemos também reverter a string usando a seguinte notação:
print(nome[::-1]) # nnamueN noV nhoJ. Neste caso, vamos percorrer o
array de traz para frente passando por cada posição (-1).
```

Métodos em Strings

Um detalhe importante dentro do universo das **strings** é que podemos utilizar **métodos** (funções especiais) para manipulá-las.

Para compreendermos melhor estes métodos, vamos definir uma string para podermos testá-los:

```
nome = " Meu nome é Dennis Ritchie "
```

O método **strip()** remove todos os espaços extras no começo e no fim da string:

```
print(nome.strip()) # Meu nome é Dennis Ritchie
```

O método **len()** retorna o tamanho (comprimento) da string:

```
print(len(nome)) # 27
```

O método **lower()** retorna a string em lower case (todos os caracteres se tornam minúsculos):

```
print(nome.lower()) # meu nome é dennis ritchie
```

O método **upper()** retorna a string em upper case (todos os caracteres se tornam maiúsculos):

```
print(nome.upper()) # MEU NOME É DENNIS RITCHIE
```

O método **swapcase()** retorna a string com caracteres uppercase convertidos para lowercase e vice-versa:

```
print(nome.swapcase()) # mEU NOME É dENNIS rITCHIE
```

O método **title()** retorna a string ao qual a primeira letra de cada palavra é convertida para uppercase e as demais letras permanecem lowercase:

```
print(nome.title()) # Meu Nome É Dennis Ritchie
```

O método **replace()** substitui a string que desejarmos por outra string especifica por nós via argumento:

- Primeiro informamos a string a ser substituída
- Segundo informados a nova string

```
print(nome.replace("Dennis", "Ken")) # Meu nome é Ken Ritchie
```

O método **split()** separa a string em substrings caso haja um separador, nesse caso utilizamos o espaço (" ") e ele nos retorna uma lista de strings:

```
print(nome.split(" ")) # ['', 'Meu', 'nome', 'é', 'Dennis', 'Ritchie', '']
```

O método **join()** retorna a string que resulta da concatenação dos objetos em um iterável (uma lista, por exemplo) separados por delimitador.

No exemplo a seguir, o delimitador é a string ',' e o iterável é uma lista de filósofos:

```
filosofos = ['Kant', 'Kierkegaard', 'Nietzsche', 'Leibniz']
print(', '.join(filosofos)) # Kant, Kierkegaard, Nietzsche, Leibniz
print(' - '.join(filosofos)) # Kant - Kierkegaard - Nietzsche - Leibniz
```

O resultado é uma única string que consiste nos objetos da lista separados por um delimitador especificado por nós.

No próximo exemplo, iterável é especificado como um único valor de string (“aeiou”). Quando um valor de string é usado como um iterável, ele é interpretado como uma lista dos caracteres individuais da string:

```
list('aeiou') # ['a', 'e', 'i', 'o', 'u']
```

Abaixo mostramos o método join sobre a string "aeiou".

```
'|'.join('aeiou') # 'a|e|i|o|u'
```

O método **count()** retorna o número de ocorrências não sobrepostas de substring informada por nós:

```
print("Ra Ra Ja Ra Ta".count("Ra")) # 3
```

O método **startwith()** retorna True se uma determinada string começa com um prefixo especificado, caso contrário, retorna False:

```
print("existencialismo".startswith("exist")) # True
print("existencialismo".startswith("ismo")) # False
```

O método **endswith()** retorna True se uma determinada string termina com um sufixo especificado, caso contrário, retorna False:

```
print("existencialismo".endswith("ismo")) # True
print("existencialismo".endswith("exist")) # False
```

O método **find()** pode ser usado para vermos se uma string contém uma substring informada e retorna o menor índice na string onde esta substring é encontrada:

```
print('Amar, é encontrar a própria felicidade na felicidade
alheia'.find('Amar')) # 0
print('Amar, é encontrar a própria felicidade na felicidade
alheia'.find('é')) # 6
```

Os métodos de classificação de caracteres são capazes de classificar uma string baseado nos caracteres contidos nela. Vejamos alguns exemplos.

O método **isalnum()** retorna True se a string for não-vazia e todos os seus caracteres forem alfanuméricos (ou uma letra ou um número), e caso contrário, retorna False.

```
print('xyz678'.isalnum()) # True
print('xyz#678'.isalnum()) # False
print(' '.isalnum()) # False
```

O método **isalpha()** retorna True se a string for não-vazia e todos os seus caracteres forem alfabéticos, caso contrário, retorna False.

```
print('exemplo'.isalpha()) # True
print('exemplo 2'.isalpha()) # False
```

Para checarmos se uma string contém apenas dígitos, podemos utilizar o método **isdigit()**, ele vai retornar True se a string for não-vazia e todos os caracteres forem numéricos, e caso contrário, retornará False.

```
print('33'.isdigit()) # True
print('a33z'.isdigit()) # False
print(''.isdigit()) # False
```

O método **isidentifier()** nos retorna True se uma determina string é um identificador válido de acordo com a definição da linguagem Python, caso contrário, retorna False.

```
print('nome'.isidentifier()) # True
print('nome2'.isidentifier()) # True
print('2nome'.isidentifier()) # False
print('nome#'.isidentifier()) # False
```

Dessa forma podemos saber como definir uma variável corretamente.

Já o método **iskeyword()** é capaz de testar se uma determinada string corresponde à uma palavra-chave Python, este método é contido no módulo **keyword**, sendo então necessário importá-lo.

```
from keyword import iskeyword
print(iskeyword("or")) # True
print(iskeyword("else")) # True
print(iskeyword("while")) # True

print(iskeyword("switch")) # False
```

O método **isprintable()** determina se uma string consiste inteiramente de caracteres imprimíveis. Ele retorna True se a string for vazia ou se todos os caracteres alfabéticos que ela contém forem imprimíveis, retorna False se ela contiver pelo menos um caractere não-imprimível. Caracteres não-alfabéticos são ignorados.

```
print('Gabriel\tFelippe'.isprintable()) # False
print('Gabriel Felippe'.isprintable()) # True
print(''.isprintable()) # True
print('x\ny'.isprintable()) # False
```

O método **isspace()** determina se uma string consiste de caracteres de **espaço em branco**, retornará True se a string for não-vazia e todos os caracteres forem de espaço em branco, caso contrário, retornará False.

```
print(''.isspace()) # False
print('x'.isspace()) # False
print(' '.isspace()) # True
print('\t\n'.isspace()) # True
```

O método **isupper()** é capaz de verificar se todos os caracteres de uma string são uppercase:

```
print("STRING".isupper()) # True
print("String".isupper()) # False
```

Já o método **islower()** é capaz de identificar se todos os caracteres de uma string são lowercase:

```
print("String".islower()) # False
print("string".islower()) # True
```

Formatando Strings

Vejamos agora como podemos usar métodos que modificam ou aprimoram o formato de uma string.

O método **center()** retorna uma string centralizada em um determinado comprimento especificado por nós via argumento, por padrão, o preenchimento consiste no caractere de **espaço** ASCII, por exemplo:

```
'python'.center(20) # '       python       '
'python'.center(25,'.') # '.........python.........'
```

O método **expandtabs()** substitui cada caracter **tab** (\t) com espaços.

```
'x\ty'.expandtabs() # 'x       y'
```

O método **ljust()** retorna uma string justificada à esquerda em um campo de comprimento especifica por nós via argumento, por padrão, o preenchimento consiste no caractere de **espaço** ASCII, por exemplo:

```
'C++'.ljust(10) # 'C++       '
'C++'.ljust(10,'-') # 'C++-------'
```

O método **rjust()** retorna uma string justificada à direita em um campo de comprimento especifica por nós via argumento, por padrão, o preenchimento consiste no caractere de **espaço** ASCII, por exemplo:

```
'Javascript'.rjust(20) # '          Javascript'
'Javascript'.rjust(20,'-') # '----------Javascript'
```

O método **zfill()** retorna uma cópia da string com preenchimento à esquerda com caracteres '0', o comprimento é especificado por nós, por exemplo:

```
'66'.zfill(4) # '0066'
```

Embora seja mais interessante para números, **zfill()** também funciona com caracteres:

```
'letra'.zfill(7) # '00letra'
```

O método **format()** nos permite construir **strings** de uma maneira flexível:

```
nome = "Thiago"

profissao = "Professor"

print("A profissão de {0} é {1}".format(nome,profissao))
# A profissao de Thiago é Professor
```

F-Strings

F-strings são um novo tipo de *string literals* introduzido na versão **3.6** do Python. A **string** formatada é prefixada com a letra f e é similar à formatação de strings aceita por format().

Vejamos exemplos:

```
first_name = "Alan"
last_name = "Turing"
sentenca = f'Meu nome e {first_name.upper()} {last_name.lower()}'
print(sentenca) # Meu nome e ALAN turing
```

Observe que podemos também executar métodos com F-strings.

Além disso, podemos também utilizar **dicionários**:

```
pessoa = {'nome': 'Muhammad', 'idade': 22}
sentenca = f'Meu nome é {pessoa["nome"]} e eu tenho {pessoa["idade"]}
anos de idade'
print(sentenca)

# Meu nome é Muhammad e eu tenho 22 anos de idade
```

Ou até mesmo realizar operações matemáticas:

```
calculo = f'4 vezes 11 é igual a {4 * 11}'
print(calculo) # 4 vezes 11 é igual a 44
for n in range(1,11):
    sentenca = f'O Valor é {n:02}'
    print(sentenca)
```

Para formatarmos datas precisaremos importar a biblioteca **datetime**:

```
from datetime import datetime
nascimento = datetime(1991, 6, 6)
sentenca = f'O Nascimento é no dia {nascimento:%B %d, %Y}'
print(sentenca)
# O Nascimento é no dia June 06, 1991
```

Convertendo para o Formato String

Para fazermos a conversão para uma string podemos utilizar os métodos repr() e str():

```
x = 10
y = [1, 3.14, 17, 13]
print(type(repr(x))) # <class 'str'>
print(type(str(y))) # <class 'str'>
```

Escape Caracteres

Lembram que eu falei dos caracteres de escape. Pois bem, agora é a hora de mostrar o que eles representam. Observem a tabela abaixo:

Carácter	Descrição
\n	nova linha
\r	carriage return
\s	espaço
\t	tab
\v	tab vertical
\e	escape
\b	backspace

Vejamos alguns exemplos dos escape caracteres:

```
>>> print("Meu nome é \nThiago")
... print("\tOlá meu amigo")
...
Meu nome é
Thiago
    Olá meu amigo
```

Operadores

Operadores Especiais:

Operador	Descrição
+	Concatenação, adiciona os valores
*	Repetição, concatena múltiplas cópias da mesma string (lembram do exemplo vídeo?)
in	Para verificarmos se determinado caractere existe na string
not in	Para verificarmos se determinado caractere não existe na string

Veja um exemplo dos operadores especiais:

```
nome = "Carl"
sobrenome = "Sagan"
print(nome +" "+ sobrenome) # Concatena o nome e sobrenome
print(nome*10)              # Multiplica o nome por 10 (o nome vai
aparecer 10 vezes)
print("C" in nome)          # Verifica se C está presente em nome e
retorna True
print("C" not in nome)      # Verifica se C não está presente em nome e
retorna False
```

É importante lembrarmos que **strings** são imutáveis, uma vez criada, não pode ser modificada. Para alterar uma **string** você deve primeiro construir uma nova string através da concatenação ou chamada de uma função e então **re-atribuir** à variável **string**.

Slice de palavras

Um assunto que a FGV gosta relacionado ao assunto de Python é o fatiamento ou slice de objetos. Vamos apresentar abaixo a lógica do slice para palavras, que se repete dentro dos demais objetos Python. Fatiamento significa extrair apenas uma parte da string, ou seja, uma substring. Com essa operação, podemos delimitar os limites inferior e superior do pedaço da string que queremos acessar. Por exemplo, se quisermos acessar a substring da posição 0 até a posição 5 na string palavra original, podemos fazer o seguinte:

```
palavra = "Python no TCU!"
palavra[0:6] #(da posição 0 até a 5)
```

Existem outras possibilidades com o fatiamento, como usar números negativos para referenciar posições e deixa o valor em branco. Se o valor em branco aparecer a direita estamos lendo a substring de uma determinada posição até o final. Se o valor em branco aparecer a esquerda dos dois pontos (:) estamos lendo do ínicio o texto até uma posição especificada pelo valor que aparece após os dois pontos.

```
palavra[-4:-1] #(da posição -4 até -2)
palavra[7:] #(da posição (7 até o final)
palavra[:6] #(da posição 0 até a 5)
```

O fatiamento de strings pode aceitar um terceiro parâmetro, além dos dois números de índice. O terceiro parâmetro especifica o stride (deslocamento), que diz respeito a quantos caracteres devem ser pulados após o primeiro caractere ser recuperado da string. Até agora, omitimos o parâmetro stride e o Python utiliza o valor padrão do stride de 1, para que todos os caracteres entre dois números de índice sejam recuperados. Vamos observar novamente o exemplo acima que imprime a substring “Aprovação na certa!”:

```
EstrategiaConcurso = “Aprovação na certa!”

EstrategiaConcurso[10:18]
'na certa'
```

Podemos obter os mesmos resultados incluindo um terceiro parâmetro com um deslocamento de 1:

```
EstrategiaConcurso [10:18:1]
'na certa'
```

Assim, um deslocamento de 1 irá abranger todos os caracteres entre dois números de índice de uma fatia. Se omitirmos o parâmetro deslocamento, então o Python usará o padrão 1. Se, ao invés disso, aumentarmos o deslocamento, veremos que alguns caracteres são ignorados:

```
EstrategiaConcurso[7:13:2]
'n cra'
```

Especificar o deslocamento de 2 como o último parâmetro na sintaxe do Python atcu[0:13:2] ignora um caractere a cada dois. Vamos ver os caracteres que são impressos em vermelho:

Aprovação na certa!

```
EstrategiaConcurso[0:13:2]
'Arvço a cra'
```

Nestes exemplos, o caractere de espaço em branco também é considerado. Como estamos imprimindo toda a string, podemos omitir os dois números de índice e manter os dois sinais de dois pontos dentro da sintaxe para alcançar o mesmo resultado:

```
EstrategiaConcurso[::4]
'Avo t'
```

Omitir os dois números de índice mantendo os dois pontos irá considerar a string inteira dentro do intervalo, ao mesmo tempo que adicionando um parâmetro final para o deslocamento especificará o número de caracteres a serem pulados.

Além disso, é possível indicar um valor numérico negativo para o stride, que podemos usar para imprimir a string original em ordem reversa se definirmos o deslocamento para -1:

```
EstrategiaConcurso [::-1]
'!atrec an oãçavorpA'
```

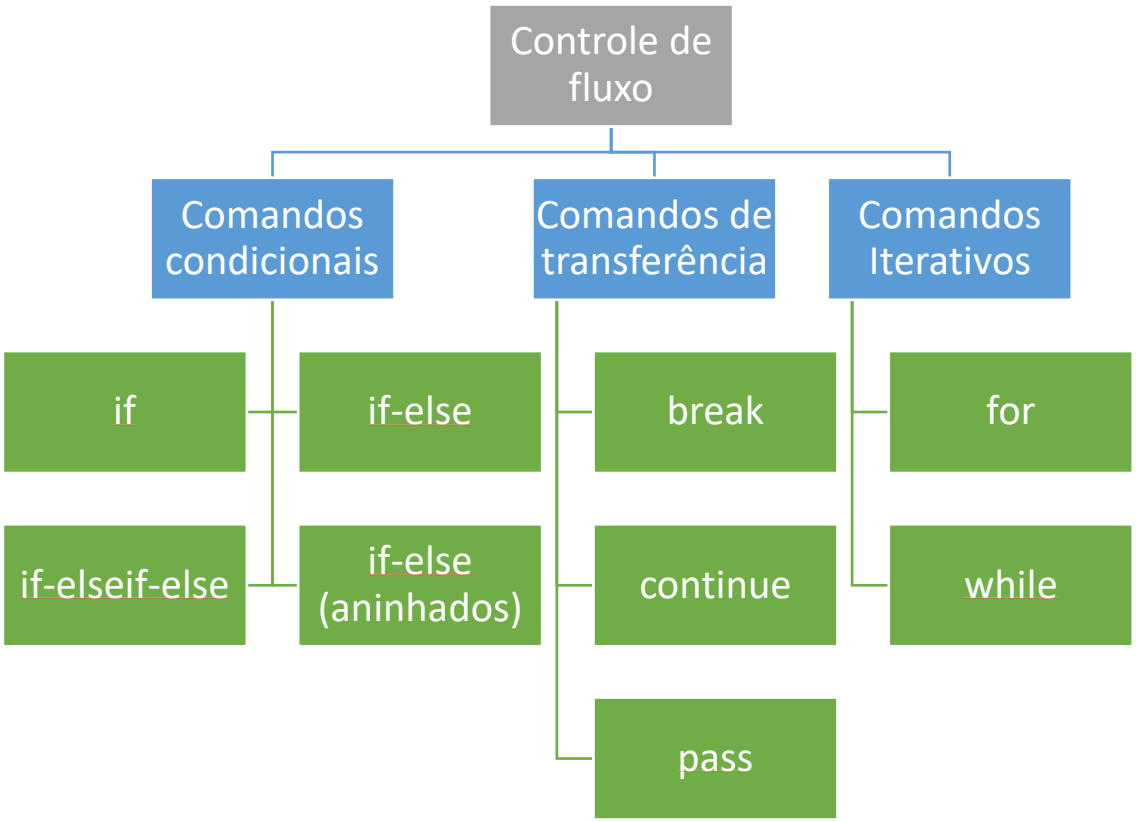
Os dois sinais de dois pontos sem parâmetro especificado incluirão todos os caracteres da string original, um deslocamento de 1 incluirá todos os caracteres sem pular nenhum, e o deslocamento negativo inverterá a ordem dos caracteres. Vamos fazer isso novamente, mas com um deslocamento de -2:

```
EstrategiaConcurso[::-2]
'!ençr'
```

Neste exemplo, `EstrategiaConcurso` [::-2], estamos lidando com a totalidade da string original, uma vez que nenhum número de índice foi incluído nos parâmetros e invertendo a string ao utilizar o deslocamento negativo. Além disso, por termos um deslocamento de -2, estamos ignorando um caractere a cada dois na string invertida:

O caractere de espaço em branco não é impresso neste exemplo. Ao especificar o terceiro parâmetro da sintaxe do *slice* do Python, você está indicando o deslocamento da substring que está sendo gerada a partir da string original.

Estruturas de controle de fluxo.



Comandos condicionais

Agora, finalmente damos nosso primeiro passo nas *estruturas de código* que deliberam o fluxo em programas. Pense assim: se temos um programa que executa várias funcionalidades, precisamos executar cada parte do código no momento exato, quando realmente precisamos dele. Nosso primeiro exemplo é este pequeno programa Python que verifica o valor da variável booleana *aprovado* e imprime um comentário apropriado:

```
>>> aprovado = True
>>> if aprovado:
```

```
...     print("Sucesso total! Vamos tomar uma para comemorar!!!")
... else:
...     print("Eita! Não desiste!! Vamos tomar uma para esquecer")
...
Sucesso total! Vamos tomar uma para comemorar!!!
```

Nas linhas, if e else são *instruções* Python que verificam se uma condição (aqui, o valor de *aprovado*) é um valor booleano ou pode ser avaliada como verdadeiro (True). Lembre-se, print() é a *função* interna do Python para imprimir coisas, normalmente na sua tela.

Se você programou em outras linguagens, observe que você não precisa de parênteses para o teste dentro do if. Por exemplo, não é necessário digitar algo como if (aprovado == True) (o operador de igualdade "==" será descrito em alguns parágrafos). Você precisa apenas dos dois pontos (:). Se você esquecer de digitar os dois pontos, o Python exibirá uma mensagem de erro.

Cada linha com comando print() é avançada em relação ao seu teste. Usei quatro espaços para recuar cada subseção. Embora você possa usar qualquer recuo que desejar, o Python espera que você seja consistente com o código dentro de uma seção - as linhas precisam ser recuadas na mesma quantidade, alinhadas à esquerda. O estilo recomendado, chamado PEP-8, é usar quatro espaços. Não use tabulações nem misture tabulações e espaços; isso atrapalha a contagem de recuo.

Já fizemos várias coisas até aqui:

- Atribuímos o valor booleano *True* à variável chamada *aprovado*
- Realizamos uma *comparação condicional* usando if e else, executando um código diferente dependendo do valor da variável.
- *Chamamos a função* print() para imprimir algum texto

Você pode ter testes dentro de testes, com quantos níveis de profundidade forem necessários vejamos outro exemplo:

```
>>> furry = True
>>> large = True
>>> if furry:
...     if large:
...         print("It's a yeti.")
...     else:
...         print("It's a cat!")
... else:
...     if large:
...         print("It's a whale!")
...     else:
...         print("It's a human. Or a hairless cat.")
...
It's a yeti.
```

Em Python, a indentação determina como as seções if e else são pareadas. Nosso primeiro teste foi verificar o valor de ***furry***. Como furry tinha o valor True, o Python vai testar o if large. Como definimos ***large*** como True, if large é avaliado como True, e a linha else a seguir é ignorada. Isso faz com que o Python execute a linha após o *if large*: e imprima *It's a yeti*.

Se houver mais de duas possibilidades para testar, use um if para a primeira, um elif (ou seja, *else if*) para as do meio e else para a última:

```
>>> color = "mauve"
>>> if color == "red":
...     print("It's a tomato")
... elif color == "green":
...     print("It's a green pepper")
... elif color == "bee purple":
...     print("I don't know what it is, but only bees can see it")
... else:
...     print("I've never heard of the color", color)
...

I've never heard of the color mauve
```

No exemplo anterior, testamos a igualdade usando o operador ==. Aqui estão os *operadores de comparação* do Python:

Igualdade	==
Desigualdade	!=
Menor que	<
Menor ou igual	<=
Maior que	>
Maior ou igual	>=

Estes retornam os valores booleanos **True** ou **False**. Vamos ver como tudo isso funciona, mas primeiro, atribua um valor a x:

```
>>> x = 7
```

Agora vamos fazer alguns testes:

```
>>> x == 5
False
>>> x == 7
True
>>> 5 < x
True
>>> x < 10
True
```

Observe que dois sinais de igual (==) são usados para *testar a igualdade*; lembre-se, um único sinal de igual (=) é o que você usa para atribuir um valor a uma variável.

Se você precisar fazer várias comparações ao mesmo tempo, use os *operadores lógicos* (ou *booleanos*) ***and***, ***or*** e ***not*** para determinar o resultado booleano final.

Os operadores lógicos têm menor *precedência* do que os pedaços de código que estão comparando. Isso significa que os pedaços são calculados primeiro e depois comparados. No

exemplo abaixo, como definimos x com o valor 7, Se verificarmos se x > 5 e x < 10, então finalmente terminamos com True and True:

```
>>> 5 < x and x < 10
True
```

Aqui estão alguns outros testes:

```
>>> 5 < x or x < 10
True
>>> 5 < x and x > 10
False
>>> 5 < x and not x > 10
True
```

Se você estiver fazendo várias comparações and com uma variável, o Python permite que você faça isso:

```
>>> 5 < x < 10
True
```

É o mesmo que 5 < x and x < 10. Você também pode escrever comparações mais longas:

```
>>> 5 < x < 10 < 999
True
```

Mas o que seria esse True? E se o elemento que estamos verificando não for um booleano? O que o Python considera True e False?

Um valor **false** não precisa necessariamente ser explicitamente um booleano False. Por exemplo, todos os valores abaixo são considerados False:

boleano	False
nulo	None

inteiro zero	0
flutuação zero	0.0
string vazia	''
lista vazia	[]
tupla vazia	()
dict vazio	{}
conjunto vazio	set()

Qualquer outra coisa é considerada True. Os programas Python usam essas definições de "veracidade" e "falsidade" para verificar estruturas de dados vazias:

```
>>> some_list = []
>>> if some_list:
...     print("There's something in here")
... else:
...     print("Hey, it's empty!")
...
Hey, it's empty!
```

Se o que você está testando é uma expressão em vez de uma simples variável, o Python avalia a expressão e retorna um resultado booleano. Então, se você digitar:

```
if color == "red":
```

Python avalia color == "red". Em nosso exemplo anterior, atribuímos a string a cor "mauve", então color == "red" é False, e o Python passa para o próximo teste:

```
elif color == "green":
```

Faça múltiplas comparações com in

Suponha que você tenha uma letra e queira saber se é uma vogal. Uma maneira seria escrever uma longa declaração if:

```
>>> letter = 'o'
>>> if letter == 'a' or letter == 'e' or letter == 'i'\
...     or letter == 'o' or letter == 'u':
...     print(letter, 'é uma vogal')
... else:
...     print(letter, 'não é uma vogal')
...
o é uma vogal
```

Sempre que você precisar fazer muitas comparações como essa, separadas por or, use o *operador de associação in* do Python. Veja como verificar a vogal de forma mais Pythônica, usando com uma string feita de caracteres de vogal:

```
>>> vogais = 'aeiou'
>>> letra = 'o'
>>> letra in vogais
True
>>> if letra in vogais:
...     print(letra, 'é uma vogal')
...
o é uma vogal
```

Aqui está uma prévia de como usar in com alguns tipos de dados:

```
>>> letter = 'o'
>>> vowel_set = {'a', 'e', 'i', 'o', 'u'}
>>> letter in vowel_set
True
>>> vowel_list = ['a', 'e', 'i', 'o', 'u']
>>> letter in vowel_list
True
>>> vowel_tuple = ('a', 'e', 'i', 'o', 'u')
>>> letter in vowel_tuple
True
>>> vowel_dict = {'a': 'apple', 'e': 'elephant',
...               'i': 'impala', 'o': 'ocelot', 'u': 'unicorn'}
>>> letter in vowel_dict
True
>>> vowel_string = "aeiou"
>>> letter in vowel_string
True
```

Para o dicionário, o ***in*** olha para as chaves (o lado esquerdo do :) em vez de seus valores.

Comandos iterativos e de transferência

Repetição com While

O mecanismo de loop mais simples em Python é while. Com loop while podemos executar um conjunto de instruções enquanto uma condição for verdadeira.

Usando o interpretador interativo, tente executar este exemplo, que é um loop simples que imprime os números de 1 a 5:

```
>>> count = 1
>>> while count <= 5:
...     print(count)
...     count += 1
...
1
2
3
4
5
>>>
```

Primeiro atribuímos o valor 1 a count. O loop while compara o valor de count com 5 e continua e execução se count for menor ou igual a 5. Dentro do loop, imprimimos o valor de count e então *incrementamos* seu valor em um com a instrução count += 1. A execução volta ao topo do loop e novamente compara count com 5. O valor de count agora é 2, então o conteúdo do loop while é executado novamente e count é incrementado para 3.

Isso continua até que count seja incrementado de 5 para 6 na parte inferior do loop. Na próxima viagem ao topo, count <= 5 é agora False, e o loop while termina. O Python segue então para as próximas linhas.

Cancelar com o break

Se você deseja fazer um loop até que algo ocorra, mas não tem certeza de quando isso pode acontecer, você pode usar um ***loop infinito*** com uma instrução break. Desta vez, vamos ler uma linha de entrada do teclado através da função input() do Python e depois imprimi-la

com a primeira letra maiúscula. Saimos do loop quando uma linha contendo apenas a letra q é digitada:

```
>>> while True:
...       stuff = input("String to capitalize [type q to quit]: ")
...       if stuff == "q":
...            break
...       print(stuff.capitalize())
...
String to capitalize [type q to quit]: test
Test
String to capitalize [type q to quit]: hey, it works
Hey, it works
String to capitalize [type q to quit]: q
>>>
```

Pular para a próxima iteração com o *continue*

Às vezes, você não quer sair de um loop, mas apenas **quer pular para a próxima iteração** por algum motivo. Aqui está um exemplo artificial: vamos ler um inteiro, imprimir seu quadrado se for ímpar e ignorá-lo se for par. Novamente, usamos q para parar o loop:

```
>>> while True:
...       value = input("Integer, please [q to quit]: ")
...       if value == 'q':       # quit
...            break
...       number = int(value)
...       if number % 2 == 0:    # an even number
...          continue
...       print(number, "squared is", number*number)
...
Integer, please [q to quit]: 1
1 squared is 1
Integer, please [q to quit]: 4
Integer, please [q to quit]: 5
5 squared is 25
Integer, please [q to quit]: q
>>>
```

Um uso específico do else junto com o break

Se o loop while terminou normalmente (sem a chamada do break), o controle passa para um else opcional. Você usa isso quando codifica um loop while para verificar algo e interrompe assim que é encontrado. O else seria executado se o loop while fosse concluído, mas o objeto não fosse encontrado:

```
>>> numbers = [1, 3, 5]
```

```
>>> position = 0
>>> while position < len(numbers):
...     number = numbers[position]
...     if number % 2 == 0:
...         print('Found even number', number)
...         break
...     position += 1
... else:  # break not called
...     print('No even number found')
...
No even number found
```

Antes de continuar vamos fazer uma rápida comparação entre o break, o continue e o else (depois de um break.

Break	Continue	Else (loop)
• A instrução break interrompe a execução de um loop for ou while (mais interno)	• A instrução continue, segue para a próxima iteração do loop	• a cláusula else de um loop é executada quando não break ocorre.

Iterar com o for e o in

Python faz uso frequente de *iteradores*, por um bom motivo. Eles possibilitam que você atravesse estruturas de dados sem saber o tamanho delas ou como elas são implementadas. Você pode até mesmo iterar sobre os dados criados em tempo real, permitindo o processamento de *fluxos* de dados que de outra forma não caberiam na memória do computador de uma só vez.

Para mostrar a iteração, precisamos de algo para iterar. Mostrarei duas maneiras de percorrer uma string aqui. O Python consegue percorrer uma string como esta:

```
>>> word = 'TCU'
>>> offset = 0
>>> while offset < len(word):
...     print(word[offset])
...     offset += 1
...
T
C
U
```

Mas há uma maneira melhor de fazer isso ... vejamos:

```
>>> for letter in word:
...     print(letter)
...
T
C
U
```

A iteração de string produz um caractere por vez.

A inserção de um *continue* em um loop for salta para a próxima iteração do loop, como acontece com um loop while.

Gerando sequências numéricas com range()

A função range() retorna um fluxo de números dentro de um intervalo especificado, sem ter que criar e armazenar uma grande estrutura de dados, como uma lista ou tupla. Isso permite que você crie intervalos enormes sem usar toda a memória do computador e travar o programa.

Estruturas de dados, funções e arquivos

Estruturas de dados

Quando falamos em estruturas de dados em Python podemos organizar os possíveis tipos de estruturas da seguinte forma:

Já falamos dos tipos de dados primitivos, em especial os numéricos e as strings. Agora vamos gastar um pouco da nossa aula para explicar cada uma das estruturas não primitivas:

List

Uma **lista** (*list*) em Python é uma sequência ou coleção ordenada de valores. Cada valor na lista é identificado por um índice. O valores que formam uma lista são chamados **elementos** ou **itens**. Listas são similares a strings, que são uma sequência de caracteres, no entanto, diferentemente de strings, os itens de uma lista podem ser de tipos diferentes.

Existem várias maneiras de se criar uma lista. A maneira mais simples é envolver os elementos da lista por colchetes ([e]).

```
[10, 20, 30, 40]
["spam", "bungee", "swallow"]
```

O primeiro exemplo é uma lista de quatro inteiros. O segundo é uma lista de três strings. Como dissemos anteriormente, os elementos de uma lista não precisam ser do mesmo tipo. A lista a seguir contém um string, um float, um inteiro e uma outra lista.

```
["oi", 2.0, 5, [10, 20]]
```

Um lista em uma outra lista é dita **aninhada** (*nested*) e a lista mais interna é chamada frequentemente de **sublista** (*sublist*). Finalmente, existe uma lista especial que não contém elemento algum. Ela é chamada de lista vazia e é denotada por [].

Tuplas

As tuplas são definidas colocando os elementos entre parênteses (()) em vez de colchetes ([]). As tuplas são imutáveis. Ou seja, tudo o que você aprendeu sobre listas - elas são ordenadas, podem conter objetos arbitrários, podem ser indexadas e fatiadas, podem ser aninhadas - vale também para tuplas. Mas elas não podem ser modificadas.

Sintaticamente, uma tupla é uma sequência de valores separadas por uma vírgula. Apesar de não ser necessário, há a convenção de se envolver uma tupla entre parêntese:

```
fernanda = ("Fernanda", "Montenegro", 1929, "Central do Brasil", 1998, "Atriz", "Rio de Janeiro, RJ")
```

Por que usar uma tupla em vez de uma lista?

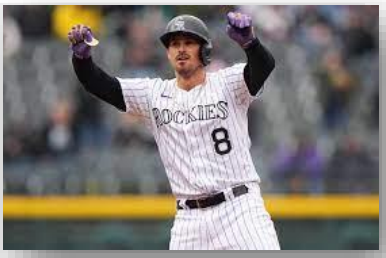
- A execução do programa é mais rápida ao manipular uma tupla do que para a lista equivalente. (Isso provavelmente não será perceptível quando a lista ou tupla for pequena.)
- Às vezes, você não quer que os dados sejam modificados. Se os valores na coleção devem permanecer constantes durante a vida do programa, usar uma tupla em vez de uma lista protege contra modificação acidental.
- Há outro tipo de dados Python que você encontrará em breve, chamado de dicionário, que requer como um de seus componentes um valor de tipo imutável. Uma tupla pode ser usada para esse propósito, enquanto uma lista não pode.

Uma das principais características de uma lista é que ela é ordenada. A ordem dos elementos em uma lista é uma propriedade intrínseca dessa lista e não muda, a menos que a própria lista seja modificada. O mesmo é verdadeiro para tuplas, exceto, é claro, que não podem ser modificadas.

Dicionário

Os elementos da lista são acessados por sua posição na lista, por meio da indexação. Os elementos do dicionário são acessados por meio de chave. Os dicionários são a implementação do Python de uma estrutura de dados que geralmente é conhecida como **array associativo**. Um dicionário consiste em uma coleção de pares de valores-chave. Cada par de valor-chave mapeia a chave para seu valor associado. Você pode definir um dicionário colocando uma lista separada por vírgulas de pares de valores-chave entre chaves ({}). Dois pontos (:) separam cada chave de seu valor associado. Vejamos um exemplo:

```
MLB_team = {
     'Colorado' : 'Rockies',
     'Boston'   : 'Red Sox',
     'Minnesota': 'Twins',
     'Milwaukee': 'Brewers',
     'Seattle'  : 'Mariners'
}
```



Conjunto

Um set em Python é uma coleção de itens únicos (distintos). Python provê formas eficientes e convenientes para criação e manipulação de sets. Vejamos algumas características e funcionalidades dessa poderosa estrutura de dados em Python.

Python nos permite criar sets de várias formas. Uma das mais formas mais frequentemente usadas é criar um set a partir de uma lista de elementos.

```
# Exemplo de criação de sets.
numeros = [1, 2, 2, 3, 3, 3]
numeros_distintos = set(numeros)
print("Números: ", numeros)
print("Números distintos: ", numeros_distintos)
Números: [1, 2, 2, 3, 3, 3]
Números distintos: {1, 2, 3}
```

Outra forma de criarmos sets em Python é criar um conjunto vazio e inserir elementos nele à medida que desejarmos. Vejamos um exemplo.

```
# Exemplo de criação de sets.
numeros = [1, 2, 2, 3, 3, 3]
numeros_distintos = set() #1
for num in numeros:
    numeros_distintos.add(num) #2
print("Números: ", numeros)
print("Números distintos: ", numeros_distintos)
Números: [1, 2, 2, 3, 3, 3]
Números distintos: {1, 2, 3}
```

#1 - Cria um conjunto vazio.

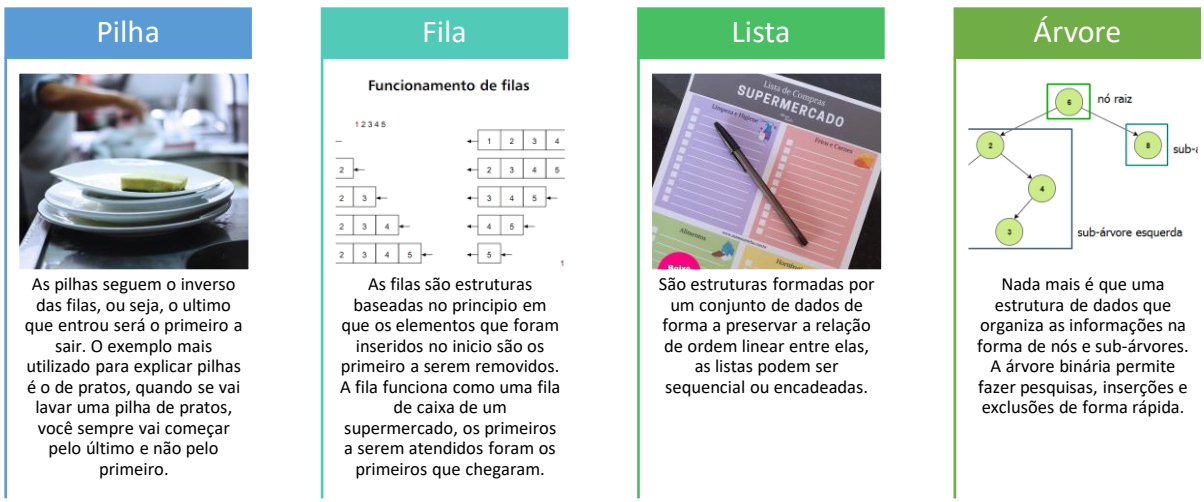
#2 - Adiciona um elemento ao conjunto criado anteriormente. Se o elemento não existir no conjunto, ele é adicionado. Caso contrário, o elemento é simplesmente descartado (não é inserido, uma vez que já está presente no conjunto).

Um ponto importante a ser discutido é a diferença de inserção de elementos em listas e conjuntos. Para inserir um elemento em uma lista, podemos usar a função insert ou a função append, mas para inserir um elemento em um set podemos usar somente a função add. Essa diferença decorre do fato de que em uma lista temos um controle da posição dos elementos: insert nos permite inserir um elemento em uma posição específica da lista e append adiciona um elemento ao final da lista. Mas em um set não temos controle sobre a ordem na qual os elementos são armazenados. A única garantia que temos é que elementos duplicados não serão inseridos.

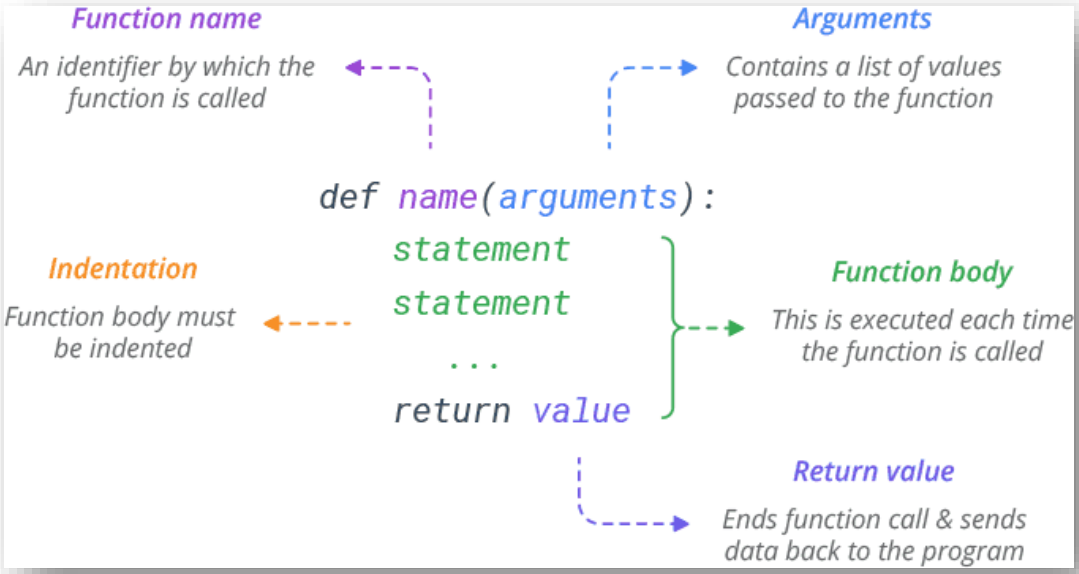
Além disso, ao percorrer uma lista, sabemos que os elementos serão percorridos na ordem em que foram armazenados nela, mas em um set não temos esse controle: dados dois sets com os mesmos elementos, ao percorrê-los é possível que a ordem dos elementos seja diferente.

Pilha, Fila, Lista ligada, Árvore

Não acho que esse tópico seja cobrado na sua prova, mas acho importante você entender os conceitos básicos de cada uma dessa estruturas de dados usadas em vários pontos da ciência da computação. Vejamos o esquema a seguir:



Funções



Até aqui vimos como criar programas simples em Python. Em todos esses programas, salvamos os resultados das computações em variáveis ou então imprimimos os resultados desejados. Entretanto, existem situações nas quais uma dada porção do nosso programa será utilizada múltiplas vezes, em diferentes partes do programa, ou até mesmo por outros programas.

Imagine, por exemplo, que desejemos criar um programa para exibir a lista dos funcionários de uma empresa, ordenada alfabeticamente, depois por idade, depois por salário. Nesse caso, temos uma tarefa (ordenar uma lista) que será executada várias vezes (primeiro para ordenar os funcionários por nome, depois por idade, e depois por salário). Em casos como esse, é conveniente que se tenha um procedimento que ordena uma lista de acordo com algum critério. Assim, esse procedimento pode ser aplicado várias vezes sem que tenhamos que repetir o mesmo código em várias partes do programa. Em outras palavras, precisamos de um procedimento que seja reutilizável em várias partes de nosso programa.

Uma das principais formas de se criar componentes reusáveis em Python (e em outras linguagens de programação também) **são as chamadas *funções***. Uma função é basicamente um trecho de código (um procedimento, um algoritmo) que pode ser usado várias vezes.

Apesar de só estarmos falando sobre funções agora, nós já usamos funções várias vezes nas seções anteriores. Por exemplo, quando fazemos x = abs(y) estamos usando a função abs() da biblioteca padrão Python para calcular o valor absoluto de y e armazená-lo em x. Agora aprenderemos a criar e usar nossas próprias funções em Python.

Funções em Python são definidas por meio da seguinte sintaxe:

```
def nomedafuncao(parametro1, parametro2, ...):
    corpodafuncao
```

Os exemplos abaixo ilustram melhor a criação de funções.

```
# Exemplo (bem simples) de função em Python.
def f(n): #1
    print("O valor do parâmetro é ", n) #2

f(10) #3

O valor do parâmetro é 10
```

#1 Nesta linha, estamos declarando uma função f que recebe um parâmetro n.

#2 O corpo da função consiste de apenas um comando, o print.

#3 Nesta linha, estamos invocando (chamando) a função f() passando o número 10 como parâmetro.

Em vez de imprimir o parâmetro, podemos simplesmente retorná-lo, como abaixo:

Exemplo de função que retorna um valor.

```
def f(n):
    return n

resultado = f(10)
print("O valor do parâmetro é {}".format(resultado))
O valor do parâmetro é 10
```

O comando ***return*** retorna um valor. Este valor deve ser salvo em alguma variável para ser usado depois, como fizemos no exemplo acima.

A função acima ilustra o uso do comando ***return***, mas ela não faz nada muito útil. Porém, não se engane: este recurso é muito útil e muito utilizado. Poderíamos, por exemplo, realizar alguma computação com o parâmetro e retornar o *resultado* desta computação. Vejamos um exemplo.

```
# Outro exemplo de função que retorna um valor.
def eleva_ao_quadrado(n):
    return n ** 2

print("O número {} elevado ao quatrado é {}".format(10,
eleva_ao_quadrado(10)))
O número 10 elevado ao quatrado é 100
```

Funções em Python podem realizar computações tão complicadas quanto desejarmos. Elas são uma ferramenta poderosa de *abstração* em programação.

Outras coisa interessante é que funções em Python podem retornar mais de um valor:

```
# Exemplo de função que retorna mais de um valor.
def quociente_resto(x, y):
    quociente = x // y
    resto = x % y
    return (quociente, resto)

print("Quociente e resto: ", quociente_resto(9, 4))
Quociente e resto:  (2, 1)
```

O exemplo acima mostra um uso interessante de tuplas: o retorno de mais de um valor em uma única chamada de função. Em geral, quando desejamos retornar mais de um valor em uma função, retornarmos uma tupla com os valores desejados. Tenha isso em mente porque essa técnica é bastante usada. E não se preocupe se você não entende o que é uma tupla. Cobriremos este tópico em breve.

Funções anônimas (funções lambda)

Para finalizar essa seção, trataremos de um tópico um pouco mais avançado: funções anônimas, funções *lambda*, ou funções de alta ordem em Python.

A palavra-chave lambda em Python nos permite criar funções anônimas. Este tipo de função é útil quando desejamos passar uma função simples como argumento para outra função.

A função ***map*** aplica uma função a um conjunto de valores. No exemplo abaixo, mostraremos como aplicar uma outra função (eleva_ao_quadrado) a todos os elementos de uma lista. Retornaremos uma nova lista contendo cada número da lista de entrada elevado ao quadrado.

```
def eleva_ao_quadrado(n):
    return n ** 2

# Função map sem o uso de função lambda.
print(list(map(eleva_ao_quadrado, range(5))))
[0, 1, 4, 9, 16]
```

Perceba que a função eleva_ao_quadrado é uma função muito simples, cujo único objetivo é ser passada como parâmetro para a função map. Podemos alcançar o mesmo resultado do código acima de forma mais concisa se usarmos uma função *lambda*.

```
# Função map com função lambda.
print(list(map(lambda x: x ** 2, range(5))))
[0, 1, 4, 9, 16]
```

Por mais simples que isso possa parecer, como programador, a maior parte do tempo você estará criando funções, então ter prática nisso ajudará muito. Vejamos um exemplo: podemos criar uma função para calcular o fatorial de um número. O exemplo abaixo mostra uma função que recebe um número como parâmetro e retorna o fatorial desse número.

```
# Exemplo mais elaborado de função em Python.

def fatorial(n):
    fat = 1
    while n > 1:
        fat *= n
        n -= 1
    return fat

print("O fatorial de {} é {}".format(6, fatorial(6)))
O fatorial de 6 é 720
```

Vejamos como a FGV já cobrou esse tópico em provas anteriores:

Arquivos

Até agora, os dados que temos usado nos programas desta aula ou estão inseridos diretamente no código ou são digitados pelo usuário à medida que são lidos. Na vida real dados estão em um arquivo. Por exemplo as imagens na unidade de processamento de imagens residem em arquivos que estão no disco rígido (*hard drive*) do seu computador. Página na Internet, documentos, músicas são outros exemplos de dados que vivem em arquivos. Nesta seção apresentaremos de maneira sucinta os conceitos de Python necessários para usar em programas dados que estão em arquivos.

Para os nossos propósitos, suporemos que os nosso arquivos de dados contém texto–isto é, arquivos de caracteres. Os programas em Python que você escreve são armazenados em arquivos de texto. Podemos criar esses arquivos de diversas maneiras. Por exemplo, podemos usar um editor de textos para digitar e salvar os dados. Podemos também baixar (*download*) os dados de uma página na Internet (*website*) e salvá-los em um arquivo. Não importa como os arquivos são criados, Python nos permitirá manipular o seu conteúdo.

Em Python, devemos **abrir** (*open*) arquivos antes de usá-los e **fechar** (*close*) os arquivos depois de que tivermos terminado de utilizá-los. Como você pode imaginar, depois de aberto um arquivo passa a ser um objeto Python de maneira semelhante que outros dados. A tabela abaixo mostra os métodos que podem ser usados para abrir e fechar arquivos.

Nome de Método	Uso	Explicação
open	open(nome_arquivo,'r')	Abre um arquivo chamado nome_arquivo e o usa para leitura. Retorna uma referência para um objeto *file*.
open	open(nome_arquivo,'w')	Abre um arquivo chamado nome_arquivo e o usa para escrita. Retorna uma referência para um objeto *file*.
close	ref_arquivo.close()	Utilização do arquivo referenciado pela variável ref_arquivo terminou.

Com essas informações já conseguimos fazer uma questão recente da FGV, vejamos:

dd

APOSTA ESTRATÉGICA

A ideia desta seção é apresentar os pontos do conteúdo que mais possuem chances de serem cobrados em prova, considerando o histórico de questões da banca em provas de nível semelhante à nossa, bem como as inovações no conteúdo, na legislação e nos entendimentos doutrinários e jurisprudenciais[3].

Resumo dos conceitos de Python

As linguagens dinâmicas eram vistas, no passado, apenas como **linguagens script**, utilizadas para automatizar pequenas tarefas. Porém, com o passar do tempo, elas cresceram, amadureceram e conquistaram seu espaço no mercado, a ponto de chamar a atenção dos grandes fornecedores de tecnologia.

Em suma, as linguagens de script são linguagens de programação que suportam scripts. Mas o que são scripts? São programas escritos para um ambiente de execução que pode interpretar (e, não, compilar) e automatizar a execução de tarefas ou instruções que poderiam ser executadas uma de cada vez por um operador humano. Por exemplo, é possível criar um script para desligar o computador automaticamente após um determinado tempo, ou ainda desabilitar e habilitar a placa de rede de forma programada.

Python é uma linguagem de programação de alto nível, interpretada, imperativa, multiparadigma, e de tipagem forte e dinâmica. Vamos entender essas características:

- *Alto nível = linguagem com nível de abstração bastante elevado, bem longe do código de máquina e mais próximo da linguagem humana.*)
- *Interpretada = seu código é executado linha a linha pelo interpretador e, depois, pelo sistema operacional, isto é, a linguagem escrita não é transformada em código de máquina, mas sim interpretada por outro programa.*

[3] Vale deixar claro que nem sempre será possível realizar uma aposta estratégica para um determinado assunto, considerando que às vezes não é viável identificar os pontos mais prováveis de serem cobrados a partir de critérios objetivos ou minimamente razoáveis.

- *Multiparadigma = suporta mais de um paradigma de programação. No caso, eles são: imperativo, procedural, funcional e orientado a objetos.*
- *Tipagem forte = o tipo de dado de um valor é do mesmo tipo da variável ao qual este valor será atribuído. Em outras palavras, se uma variável é do tipo inteiro, eu não posso atribuí-la um valor float.*
- *Tipagem dinâmica = verifica-se o tipo de um dado em tempo de execução e, não, em tempo de compilação.*

Python ainda é uma linguagem que suporta a grande maioria das técnicas de programação orientada a objetos, além de realizar tratamento de exceções. Tudo na linguagem Python é um objeto e sua sintaxe não é baseada diretamente em qualquer linguagem usada comumente. Em vez de vetores, ela inclui três tipos de estrutura de dados: listas, listas imutáveis e hashes, que são chamados de dicionários.

Os programas desenvolvidos em Python são automaticamente compilados para o formato portável chamado de bytecode, essa característica faz com que esses programas utilizem a linguagem básica e a biblioteca padrão e assim possam executar da mesma forma no Linux, Windows e outros sistemas operacionais apenas dependendo de um interpretador Python.

Especificações

- Funcionalidades para expressões regulares; sockets; threads; funcionalidade para data/tempo; analisadores XML; analisadores para arquivos de configuração; funcionalidades para manipulação de arquivos e diretórios; persistência de dados; capacidade para unidades de testes; bibliotecas clientes para os protocolos HTTP, FTP, IMAP, SMTP e NNTP.
- Distribuição sob uma licença própria (compatível com a GPL), que impõe poucas restrições. É permitida a distribuição, comercial ou não, tanto da linguagem quanto de aplicações desenvolvidas nela, em formato binário ou código fonte, bastando cumprir a exigência de manter o aviso de Copyright da PSF (Python Software Foundation).
- Permite a definição de funções por meio da palavra-chave **def**, seguida do nome da função e parênteses (Exemplo: **def NomeFuncao(parametrosFuncao)**). A endentação também é muito importante, visto que – diferente de outras linguagens que possuem blocos limitados por chaves (**{ }**) ou palavras-chaves (**begin/end**) – em Python blocos são delimitados pela própria endentação/tabulação.

> Em Python, o código é agrupado através da endentação, ou seja, a endentação vai dizer se uma instrução está dentro de um bloco ou de outro.

- Requer uma endentação padronizada. Em outras linguagens, como C/C++ ou JavaScript, a endentação não é necessária devido aos delimitadores de blocos, sendo utilizada somente para melhor visualização. As IDEs que suportam Python têm, em geral, a função de endentação automática, visto que é um pouco complicado ficar endentando tudo na mão.

Operador	Descrição

+	soma
-	subtração
*	multiplicação
/	divisão
//	divisão de inteiros
**	potenciação
%	módulo (resto da divisão)

Operadores Aritméticos

Operador	Descrição
==	igual
!=	diferente
>	maior
<	menor
>=	maior ou igual
<=	menor ou igual
not	negação
and	conjunção
or	disjunção

Operadores Lógicos

- Os tipos primitivos de Python podem ser simples ou compostos. Os Tipos Simples são: int (boolean), long, float e complex. Os Tipos Compostos são list, tuple, string e dictionary. Para leitura de dados, a linguagem pode utilizar a função embutida raw_input( ), que lê uma string do usuário e armazena em uma variável – seria similar a função scanf( ) da linguagem C.
- O tratamento de exceções é realizado por meio do bloco try-except. Vamos ver um pequeno exemplo de divisão por zero:

```
(x,y) = (5,0)
try:
        z = x/y
except ZeroDivisionError:
        print "Divisão por Zero!"
```

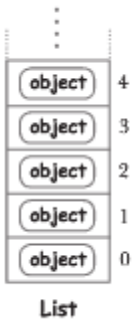
Quando se utiliza métodos em Python, convenciona-se nomear o primeiro argumento como self (Ex: def metodoQualquer(self, xpto1, xpto2)). Além disso, quando uma classe é instanciada como um objeto, o método construtor (__init__) é opcionalmente invocado. Diferente de outras linguagens, é possível acessar atributos da instância diretamente, sem precisar de métodos.

O interpretador Python executa o código de cima para baixo, uma linha de cada vez. Não é necessário terminar as linhas com ‘;’ em Python e não há a noção de uma função main()

como em outras linguagens. Além disso, não é necessário definir os tipos das variáveis e permite a definição de tipos dinâmica.

Listas

Um grande destaque do Python são as listas. As listas são objetos mutáveis e existem diversas formas de trabalhar com listas em Python.



As listas se parecem muito com vetores (arrays), porém elas podem crescer (ou diminuir) de acordo com a necessidade do sistema, trazendo maior flexibilidade.

Por exemplo, vamos criar uma lista que conterá as vogais:

```
vogais = ['a', 'e', 'i', 'o', 'u']
```

Agora vamos supor que queremos saber se alguma palavra contém vogais.

```
palavra = "Magali"
```

Antes de mais nada, perceba que palavra é do tipo String, que no fundo é também uma lista de caracteres. Vejamos:

```
for letra in palavra:
    if letra in vogais:
        print(letra)
```

A saida desse código será: **a a i**

O operador **in** serve para iterar sobre todos os elementos da lista. Ou seja, o código irá verificar para cada letra na palavra “Magali” se ela está na lista de vogais.

Funções

Funções são a pedra fundamental de programas Python.

Um conjunto de funções formam módulos e um conjunto de módulos forma a biblioteca padrão. As funções são identificadas através da palavra reservada. E caso haja algum valor de retorno, a palavra reservada é return. Veja o exemplo abaixo.

```
def nome_descritivo (argumentos_opcionais) :
    """String de
    documentação"""
    # código da função aqui
    return valor_opcional
```

As questões de Python são em sua maioria de análise de código, isso é bom, porém é importante estar familiarizado com a sintaxe.

Imprima o capítulo Aposta Estratégica separadamente e dedique um tempo para absolver tudo o que está destacado nessas duas páginas. Caso tenha alguma dúvida, volte ao Roteiro de Revisão e Pontos do Assunto que Merecem Destaque. Se ainda assim restar alguma dúvida, não hesite em me perguntar no fórum.

QUESTÕES ESTRATÉGICAS

Nesta seção, apresentamos e comentamos uma amostra de questões objetivas selecionadas estrategicamente: são questões com nível de dificuldade semelhante ao que você deve esperar para a sua prova e que, em conjunto, abordam os principais pontos do assunto.

A ideia, aqui, não é que você fixe o conteúdo por meio de uma bateria extensa de questões, mas que você faça uma boa revisão global do assunto a partir de, relativamente, poucas questões.

1.

Considere os seguintes operadores:

Exponenciação

Comparação de igualdade

Módulo (resto da divisão)

Assinale a lista dos símbolos que, respectivamente, representam esses operadores no Python.

A ^ == mod

B ** == %

C ** = %

D ** = mod

E ^ = mod

Comentário: Não há muito o que explicar aqui. É pura definição da linguagem:

Exponenciação (**). ex.: 5**3 = 125

Comparação de Igualdade (==). ex.: 5 == 3 (false)

Módulo ou Resto da Divisão (%). ex.: 5 mod 3 = 2

Portanto, **Letra B** é a resposta correta. Todas as outras usam operadores errados.

Gabarito: B

2.

Analise o código Python a seguir.

```
x = [1,2,3,4,5]
print (x[-1])
```

Assinale a opção que indica a saída produzida pela execução desse código.

A [1,2,3,4,5]

B 1

C [5,1]

D 5

E [5,4,3,2,1]

Comentário: A questão usa a ideia de posição, mas, ao invés do elemento ser uma string, estamos fazendo uma impressão da posição -1 de uma lista. Veja que a posição –1 é a ocupada pelo primeiro elemento a direita da lista. Neste caso, temos o valor 5. Desta forma, nosso gabarito pode ser encontrado na alternativa D.

Gabarito: D

3.

Analise o código Python a seguir.

```
x = [1,2,3,4,5]
print (x[::-1])
```

Assinale a opção que indica a saída produzida pela execução desse código.

A [1,2,3,4,5]

B 1

C [5,1]

D 5

E [5,4,3,2,1]

Comentário: Essa questão apresenta um problema de slice aplicado ao elementos lista. Veja que o x é uma lista com 5 elementos. Para acessar todos os elementos de uma lista usando o deslocamento de -1, precisamos percorrer os elementos de traz para frente. Assim, podemos inverter os elementos da lista encontramos o resultado na alternativa E.

Gabarito: E

4.

Assinale a opção que indica o comando Python que produz [-2, -4].

A print(range(0, -6, 2))

B print(range(0, -4, -2))

C print(range(-2, -4, -2))

D print(range(-2, -4, 2))

E print(range(-2, -6, -2))

Comentário: O funcionamento da função range, tanto na versão 2.7 quanto nas versões 3.x:

- range(X) - retorna os números de 0 até X (sem incluir o X), incrementando de 1 em 1;
- range(Y, X) - retorna os números de Y até X (sem incluir o X), incrementando de 1 em 1;
- range(Y, X, W) - retorna os números de Y até X (sem incluir X), incrementando de W em W;

Portanto:

a) print(range(0, -6, 2)) -

ERRADO. O intervalo é vazio, pois não existe número entre 0 e -6 de +2 em +2.

b) print(range(0, -4, -2))

ERRADO. Gera o intervalo [0, -2]

c) print(range(-2, -4, -2))

ERRADO. Gera o intervalo [-2]

d) print(range(-2, -4, 2))

ERRADO. O intervalo é vazio, pois não existe número entre -2 e -4 de +2 em +2.

e) print(range(-2, -6, -2))

CERTO. Gera o intervalo [-2, -4].

Portanto, **Letra E** está correta.

Gabarito: E

5.

Analise o código Python a seguir.

```
L=[10, 12, 14, 16]
for k in range(4, -5,-1):
print(L[k])
```

Esse programa causa

A erro de sintaxe.

B erro de execução.

C a exibição de 4 valores, 16,14,12,10, nessa ordem.

D a exibição de 8 valores, 16,14,12,10,16,14,12,10, nessa ordem.

E a exibição do valor 16, somente.

Comentário: Essa questão apresenta 2 problemas. Primeiramente, a indentação da terceira linha (print()) não está avançada, logo, teriamos uma erro sintático. Outro erro que aconteceria após esse seria do erro de execução. Perceba que a lista L não possui um elementos na posição -5 nem na posição 4. Assim, teríamos nosso gabarito na alternativa A. Se consertamos esse dois erros ficaríamos com o seguinte código:

```
L=[10, 12, 14, 16]
for k in range(3, -4,-1):
print(L[k])
```

Que geraria a seguinte saída:

10 12 14 16 10 12 14 16

Gabarito: A

6.

Analise o código Python a seguir.

```
L1 = [10,12,13,14,15,18,20]
L2 = [10,12,14,18,21]
for k in range(-1, -len(L1), -1):
    if L1[k] in L2[-5:-2]:
```

print L1[k]

O comando de execução desse código produz

A erro de sintaxe.

B erro de execução.

C a exibição do número 12 somente.

D a exibição de dois números, 14 e 12, nessa ordem.

E a exibição de três números, 18, 12 e 10, nessa ordem.

Comentário: Para resolver essa questão você precisa saber 4 coisas:

- Na linha 3, a função len(L1) retorna o tamanho (length) de L1, ou seja 7.
- Ainda na linha 3, range(-1, -7, -1) gera os números de -1 a -7 (sem incluir o -7), de -1 em -1, ou seja, [-1, -2, -3, -4, -5, -6]
- Na linha 4 e 5, o índice negativo em Python faz com que os elementos da lista sejam acessados de trás para frente. Ou seja, L[-1] é o último elemento, L[-2] o penúltimo, e assim por diante.
- Na linha 4, L2[-5:-2] vai gerar uma sub-coleção de L2 que começa no índice L[-5] e vai até L[-2] (sem incluir o L[-2]), ou seja, [L[-5], L[-4], L[-3]], substituindo, [10, 12, 14].

Com base nessas considerações, analisamos o condicional na linha 4:

L[-1] = 20 --> não está em [10, 12, 14].

L[-2] = 18 --> não está em [10, 12, 14].

L[-3] = 15 --> não está em [10, 12, 14].

L[-4] = 14 --> ESTÁ em [10, 12, 14].

L[-5] = 13 --> não está em [10, 12, 14].

L[-6] = 12 --> ESTÁ em [10, 12, 14].

Assim, são exibidos os números 14 e 12, nessa ordem, **Letra D**.

Gabarito: D

7.

Considere a seguinte definição da função f, declarada na sintaxe Python.

```
def f(n):
    if n < 3:
        return n-1
    else:
```

```
        return f(n-2) + f(n-1)
print f(10)
```

Assinale o valor produzido pela execução do código acima.

A 13

B 17

C 21

D 34

E 55

Comentário: f(x) retorna n-1 se n<3, e f(n-2) + f(n-1) caso contrário.

f(10) = f(8) + f(9) = f(8) + f(7) + f(8) =

= 2*f(8) + f(7) = 2*f(6) + 2*f(7) + f(7) =

= 3*f(7) + 2*f(6) = 3*f(5) + 3*f(6) + 2*f(6) =

= 5*f(6) + 3*f(5) = 5*f(4) + 5*f(5) + 3*f(5) =

= 8*f(5) + 5*f(4) = 8*f(3) + 8*f(4) + 5*f(4) =

= 13*f(4) + 8*f(3) = 13*f(2) + 13*f(3) + 8*f(3) =

= 21*f(3) + 13*f(2) = 21*f(1) + 21*f(2) + 13*f(2) =

= 34*f(2) + 21*f(1)

Porem f(1) = 0, e f(2) = 1. Logo:

f(10) = 34*1 + 21*0 = 34 (**Letra D**)

Gabarito: D.

8.

Analise o código Python a seguir.

```
def xpto (n1, n2):
    while n1 != n2:
        if (n1 < n2):
            n2 = n2 - n1
        else:
            n1 = n1 - n2
    return n1
print(xpto(50,5))
```

O valor exibido pelo comando print é:

A 0

B 1

C 5

D 10

E 50

Comentário: A questão apresenta um código em python e pede que se marque a saída para:

print xpto(50,5)

Primeiro, vamos vem a definição de xpto. Ele recebe (n1,n2), representados pelo 50 e 5 respectivamente. Sendo assim, devemos substituir esses dois valores nas linhas de comando do código.

Perceba que foi utilizada a estrutura de loop while (enquanto), combinada com um if. Logo, a estrutura irá se repetir até que a condição seja atendida. Por isso, teremos vários loopings até chegar no resultado pretendido. Veja abaixo:

```
while n1 != n2: //Enquanto n1 é diferente de n2, execute o if
        if (n1 < n2):
            n2 = n2 - n1
        else:
            n1 = n1 - n2 // Todos os loopings vão executar essa parte, já que não irá haver o caso descrito na primeira parte do if.
    return n1// Somente quando a execução sair do looping, será dada a saída com o novo n1. Essa será a nossa resposta!
```

1 iteração:

50-5 =45

2 iteração

45-5=40

Isso irá continuar até que n1 receba 5, já que dessa maneira n1 será igual a n2, o que irá quebrar a condição While (n1 ser diferente de n2).

Com isso, marcamos letra C (5)

Gabarito: C

9.

Considere o código Python, versão, na qual o comando print não requer parênteses.

A execução desse código:

def teste(n):

```
    for k in range(1, n+1):
        yiekd k
for x in teste (10):
    print(x)
```

A não tem efeito, pois nenhum comando print é acionado;

B provoca a exibição do número 10 na saída;

C provoca a exibição dos números de 1 até 10 na saída;

D provoca um erro de compilação;

E provoca um erro de execução.

Comentário: A questão trata da linguagem python e cobra especificamente a função "range()". Vamos examinar o código:

```
def teste (n):
    for k in range (1, n+1):
        yield k # esse yield substitui o return e serve para juntar as respostas das diversas iterações do loop (for)
for x in teste (10):
    print x
```

Aqui, o candidato deve saber a sintaxe da função range. Nessa função, primeiro número marca o início de uma lista. O segundo marca a parada da lista e não é contabilizado na saída. Sendo assim, atribui-se para n o valor 10. Com isso, o intervalo x varia de 1 até 10(1,11), já que o stop está em 11.

Logo, será impresso na saída os valores de 1 até 10, nos levando a resposta na alternativa C.

Gabarito: C.

10.

Assinale o código Python que cria um novo arquivo, contendo uma linha.

A

```
f = open("teste.txt","w")
f.write("Linha de teste\n")
f.close()
```

B

```
f = new file
f.open("teste.txt")
f.write("Linha de teste" + "\n")
f.quit()
```

C
```
f = create_file("teste.txt")
write(f, "Linha de teste" + "\n")
close(f)
```

D
```
f = system.file.io("teste.txt","output")
write(f, "Linha de teste\n")
quit(f)
```

E
```
f = textfile.new("teste")
f.write("Linha de teste" + "\n")
f.save()
```

Comentário: A função que abre um arquivo em Python é a função open(), quem têm 2 assinaturas principais (os outros parâmetros de abertura nunca foram cobrados):

open(filename): abre um arquivo de nome "filename" no modo leitura.

open(filename, mode): abre um arquivo de nome "filename" no modo indicado. Os modos são:

r - leitura (read)

w - escrita (write), sobrescreve todo conteúdo pré-existente.

x - criação exclusiva (falha se o arquivo já existir)

a - escrita (append), adiciona no final, mantendo o conteúdo pré-existente.

b - modo binário

t - modo texto (modo padrão)

Character	Meaning
`'r'`	open for reading (default)
`'w'`	open for writing, truncating the file first
`'x'`	open for exclusive creation, failing if the file already exists
`'a'`	open for writing, appending to the end of the file if it exists
`'b'`	binary mode
`'t'`	text mode (default)
`'+'`	open for updating (reading and writing)

Para escrever, a função utilizada é f.write('alguma coisa'). Portanto, gabarito Letra A.

Gabarito: A

QUESTIONÁRIO DE REVISÃO E APERFEIÇOAMENTO

A ideia do questionário é elevar o nível da sua compreensão no assunto e, ao mesmo tempo, proporcionar uma outra forma de revisão de pontos importantes do conteúdo, a partir de perguntas que exigem respostas subjetivas.

São questões um pouco mais desafiadoras, porque a redação de seu enunciado não ajuda na sua resolução, como ocorre nas clássicas questões objetivas.

O objetivo é que você realize uma autoexplicação mental de alguns pontos do conteúdo, para consolidar melhor o que aprendeu ;)

Além disso, as questões objetivas, em regra, abordam pontos isolados de um dado assunto. Assim, ao resolver várias questões objetivas, o candidato acaba memorizando pontos isolados do conteúdo, mas muitas vezes acaba não entendendo como esses pontos se conectam.

Assim, no questionário, buscaremos trazer também situações que ajudem você a conectar melhor os diversos pontos do conteúdo, na medida do possível.

É importante frisar que não estamos adentrando em um nível de profundidade maior que o exigido na sua prova, mas apenas permitindo que você compreenda melhor o assunto de modo a facilitar a resolução de questões objetivas típicas de concursos, ok?

Nosso compromisso é proporcionar a você uma revisão de alto nível!

Vamos ao nosso questionário:

Perguntas

1. **Quais são as principais características do Python?**
2. **Quais são os benefícios de usar o Python?**
3. **O que são namespaces Python?**
4. **O que são compreensões de Dict e List?**
5. **Quais são os tipos de dados internos comuns no Python?**
6. **Qual é a diferença entre arquivos .py e .pyc?**
7. **O que é fatiamento em Python?**

Perguntas com respostas

1. **Quais são as principais características do Python?**

- Python é uma linguagem **interpretada**. Isso significa que, ao contrário de linguagens como *C* e suas variantes, o Python **não precisa ser compilado** antes de ser executado. Outras linguagens interpretadas incluem *PHP* e *Ruby*.
- Python é **dinamicamente tipada**, isso significa que você não precisa declarar os tipos de variáveis quando você os declarar. Você pode fazer coisas como x = 111e, em seguida, x="Eu sou uma cadeia de caracteres" sem erro
- Python é bem adequado para ***programação orientada a objetos***, pois permite a definição de classes, juntamente com composição e herança. Python não tem especificadores de acesso (como público e privado de Java e C++).
- Em Python, **as funções** são **objetos de primeira classe**. Isso significa que eles podem ser atribuídos a variáveis, retornadas de outras funções e passados para funções. Classes também são objetos.
- **Escrever código Python é rápido,** mas executá-lo é muitas vezes mais lento do que em linguagens compiladas. Felizmente, o Python permite a inclusão de extensões baseadas em C para que os gargalos possam ser otimizados. O pacote *numpy* é um bom exemplo disso, é realmente muito rápido porque muito do processamento de números que ele faz não é realmente feito pelo Python
- Python encontra **uso em muitas áreas**- aplicações web, automação, modelagem científica, aplicações de big data e muito mais. Também é frequentemente usado como código de "cola" para fazer com que outras linguagens e componentes funcionem bem de forma integrada.

2. **Quais são os benefícios de usar o Python?**

Os benefícios do uso de python são-

- Fácil de usar – Python é uma linguagem de programação de alto nível que é fácil **de usar**, ler, escrever e aprender.
- **Linguagem interpretada** – Como o python é uma linguagem interpretada, ele executa o código linha por linha e para se ocorrer um erro em qualquer linha.
- **Dinamicamente** tipado – o desenvolvedor não atribui tipos de dados a variáveis no momento da codificação. Ele é atribuído automaticamente durante a execução.
- **Livre e de código aberto** – Python é livre para usar e distribuir. É de código aberto.
- **Suporte extensivo para bibliotecas** – Python tem vastas bibliotecas que contêm quase todas as funções necessárias. Ele também fornece a facilidade de importar outros pacotes usando o Python Package Manager (pip).
- **Portátil** – Os programas Python podem ser executados em qualquer plataforma sem exigir qualquer alteração.
- As estruturas de dados usadas em python são fáceis de usar.
- Ele fornece mais funcionalidade com menos codificação.

3. O que são namespaces Python?

Um namespace em python refere-se ao nome que é atribuído a cada objeto em python. Os objetos são variáveis e funções. À medida que cada objeto é criado, seu nome, juntamente com o espaço (o endereço da função externa na qual o objeto está), é criado. Os namespaces são mantidos em python como um dicionário, onde a chave é o namespace e o valor é o endereço do objeto. Existem 4 tipos de namespace em python-

- **Namespace interno** – Esses namespaces contêm todos os objetos internos em python e estão disponíveis sempre que o python estiver em execução.
- **Namespace global** – São namespaces para todos os objetos criados no nível do programa principal.
- **Namespaces enclosing** – Esses namespaces estão no nível superior ou na função externa.
- **Namespaces locais** – Esses namespaces estão na função local ou interna.

4. O que são compreensões de Dict e List?

Compreensões de dicionário e lista são apenas outra maneira concisa de definir dicionários e listas.

Exemplo de compreensão de lista é-

```
x=[i for i in range(5)]
```

O código acima cria uma lista como abaixo-

```
4
[0,1,2,3,4]
```

Exemplo de compreensão de dicionário é-

```
x = [i : i+2 for i in range(5)]
```

O código acima cria uma lista como abaixo-

```
[0: 2, 1: 3, 2: 4, 3: 5, 4: 6]
```

5. Quais são os tipos de dados internos comuns no Python?

Os tipos de dados internos comuns em python são-

Números – Incluem inteiros, números de ponto flutuante e números complexos. Eg.1, 7.9,3+4i

Lista – Uma sequência ordenada de itens é chamada de lista. Os elementos de uma lista podem pertencer a diferentes tipos de dados. Eg.[5,'market',2.4]

Tupla – É também uma sequência ordenada de elementos. Ao contrário das listas, as tuplas são imutáveis, o que significa que não podem ser alteradas. Eg.(3,'tool',1)

String– Uma sequência de caracteres é chamada de string. Eles são declarados entre aspas simples ou duplas. Por exemplo, etc."Sana"'She is going to the market'

Set– Conjuntos são uma coleção de itens exclusivos que não estão em ordem. Eg.{7,6,8}

Dicionário– Um dicionário armazena valores em pares de chave e valor onde cada valor pode ser acessado por meio de sua chave. A ordem dos itens não é importante. Eg.{1:'apple',2:'mango}

Booleano – Existem 2 valores booleanos - **Verdadeiro**e**Falso**.

6. **Qual é a diferença entre arquivos .py e .pyc?**

Os arquivos .py são os arquivos de código-fonte python. Enquanto os arquivos .pyc contêm o bytecode dos arquivos python. Os arquivos .pyc são criados quando o código é importado de alguma outra fonte. O interpretador converte os arquivos de .py de origem em arquivos .pyc, o que ajuda economizando tempo.

7. **O que é fatiamento em Python?**

O fatiamento é usado para acessar partes de sequências como listas, tuplas e cadeias de caracteres. A sintaxe do fatiamento é [start:end:step]. O step (passo) também pode ser omitido. Quando escrevemos [start:end], isso retorna todos os elementos da sequência desde o início (inclusive) até o elemento final - 1. Se o elemento inicial ou final for negativo, significa o i-ésimo elemento do fim para o início. O step indica o salto ou quantos elementos devem ser ignorados. Por exemplo, se houver uma lista - [1,2,3,4,5,6,7,8]. Em seguida [-1:2:2], retornará elementos a partir do último elemento até o terceiro elemento [8,6,4], pulando um elemento para que o step (passo) seja de 2.

...

Forte abraço e bons estudos.

"Hoje, o 'Eu não sei', se tornou o 'Eu ainda não sei'"

(Bill Gates)

Thiago Cavalcanti

Face: www.facebook.com/profthiagocavalcanti
Insta: www.instagram.com/prof.thiago.cavalcanti
YouTube: youtube.com/profthiagocavalcanti

ESSA LEI TODO MUNDO CONHECE:
PIRATARIA É CRIME.

Mas é sempre bom revisar o porquê e como você pode ser prejudicado com essa prática.

1. 

Professor investe seu tempo para elaborar os cursos e o site os coloca à venda.

2. 

Pirata divulga ilicitamente (grupos de rateio), utilizando-se do anonimato, nomes falsos ou laranjas (geralmente o pirata se anuncia como formador de "grupos solidários" de rateio que não visam lucro).

3. 

Pirata cria alunos fake praticando falsidade ideológica, comprando cursos do site em nome de pessoas aleatórias (usando nome, CPF, endereço e telefone de terceiros sem autorização).

4. 

Pirata compra, muitas vezes, clonando cartões de crédito (por vezes o sistema anti-fraude não consegue identificar o golpe a tempo).

5. 

Pirata fere os Termos de Uso, adultera as aulas e retira a identificação dos arquivos PDF (justamente porque a atividade é ilegal e ele não quer que seus fakes sejam identificados).

6. 

Pirata revende as aulas protegidas por direitos autorais, praticando concorrência desleal e em flagrante desrespeito à Lei de Direitos Autorais (Lei 9.610/98).

7. 

Concurseiro(a) desinformado participa de rateio, achando que nada disso está acontecendo e esperando se tornar servidor público para exigir o cumprimento das leis.

8. 

O professor que elaborou o curso não ganha nada, o site não recebe nada, e a pessoa que praticou todos os ilícitos anteriores (pirata) fica com o lucro.

Deixando de lado esse mar de sujeira, aproveitamos para agradecer a todos que adquirem os cursos honestamente e permitem que o site continue existindo.