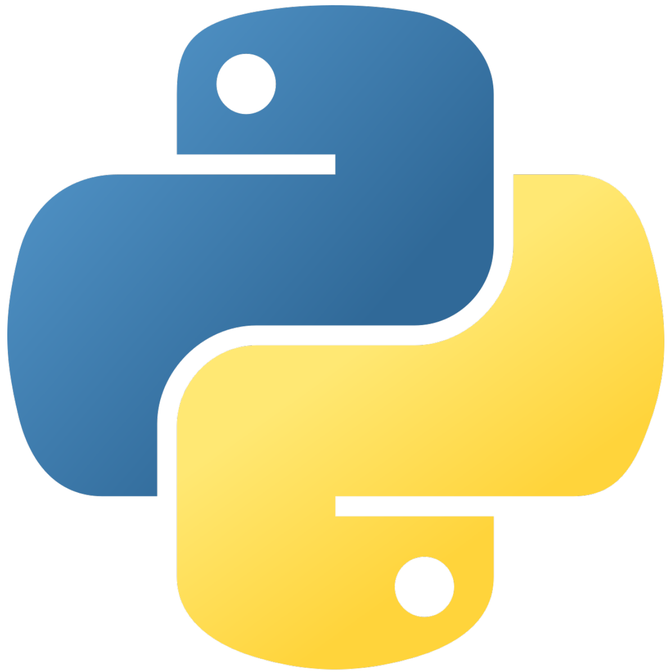
Linguagens de Programação

Módulo 7 - Estruturas Dinâmicas

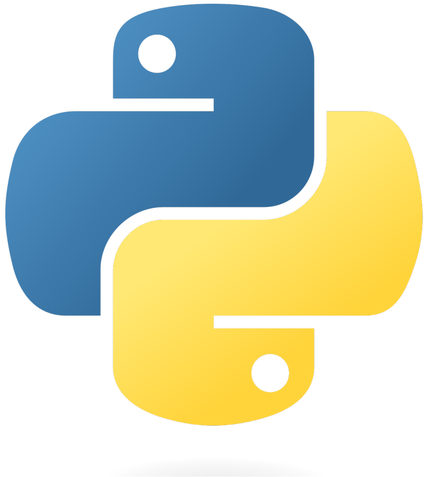


1º ano de Informática de Gestão
Escola Profissional Mariana Seixas

Professor: Nuno Castro

Linguagens de Programação

Módulo 7 - Estruturas Dinâmicas



Conteúdo:

- Gestão de memória de um computador
- Introdução a ponteiros e referências
- Alocação dinâmica de memória (heap/stack)
- Compilação / Interpretação de código
- Estruturas dinâmicas: Filas, Pilhas, Listas Ligadas, Arvores Binárias

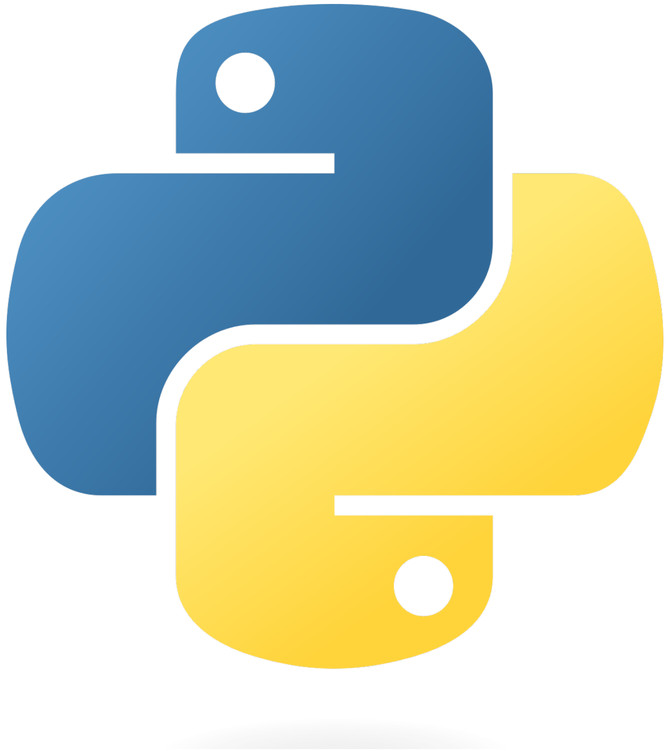
Recursos

- Documentação Python: https://docs.python.org/3/

- Editor online: https://www.onlinegdb.com

Linguagens de Programação

Módulo 7 - Estruturas Dinâmicas

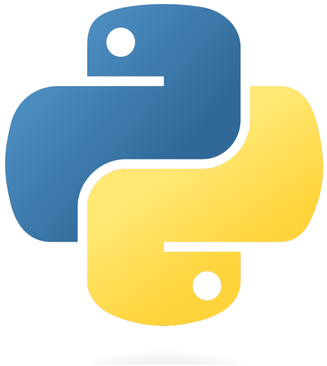


Gestão de Memória

(matéria)

Python

Módulo 7 - Estruturas Dinâmicas



- O que acontece num computador quando, por exemplo, criamos uma variável com uma linguagem de computador?
- Como é que o computador consegue converter as instruções das linguagens de programação e os dados em ações?

Memória de um computador

- O funcionamento de um computador só é possível porque existe um dispositivo ou sistema onde se pode guardar informação.
  A isso chamamos de memória.

- Um computador usa o espaço que tem nas suas diferentes memórias para realizar operações, processar e guardar informação.

- Todo o sistema funciona em sintonia, com o processador a comunicar com as diversas formas de memória que um computador tem.

- No entanto, a capacidade de processamento de um processador é muito mais veloz do que a velocidade média de armazenar a informação.

- Para que o processador não fique à espera e para optimizar a capacidade da memória em dar resposta à velocidade de processamento, a memória existe em diversas formas e representam diferentes níveis lógicos do funcionamento de um computador.

Memória de um computador

- Basicamente podemos agrupar a memória de um computador em três grupos: Cache, primária e secundária.
- A diferença entre estes grupos está na velocidade de acesso à informação e no custo que cada uma tem. Quanto mais rápido o acesso à informação for, mais dispendiosa é a memória. Quanto mais dispendiosa mais rara ela é num computador.

Memória de um computador

Cache e registos de processador

- A memória Cache é um dispositivo (semicondutor) que está, fisicamente, na mesma unidade que o processador de um computador.
- É extremamente rápida e extremamente dispendiosa.
- Um processador tem uma capacidade muito reduzida desta memória e esta serve como uma “almofada” entre o processador e a memória primária.
- Normalmente é usada para guardar, temporariamente, a informação de um programa que ou diz respeito diretamente ao processador ou que está a ser processada naquele preciso momento. É a memória interna do processador.
- Nos processadores atuais, a memória cache consegue acompanhar a velocidade de processamento de um processador.

Memória de um computador

Memória primária

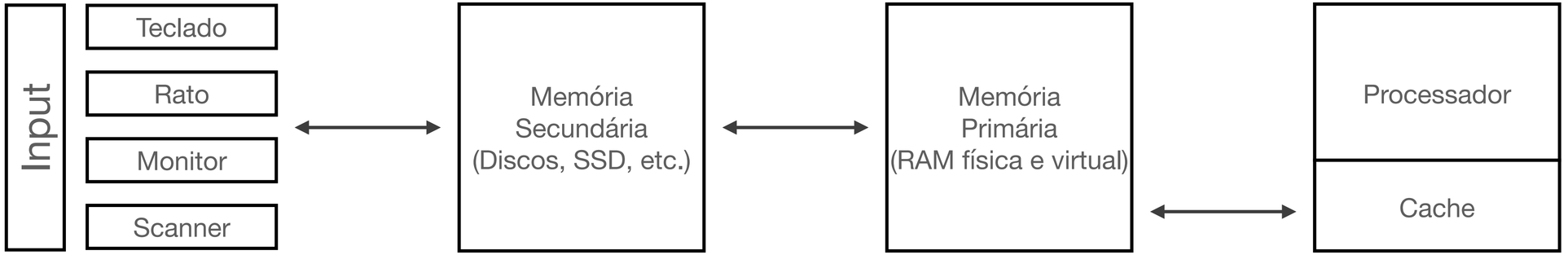
- A memória primária retém a informação e instruções dos programas que estão a ser usados pelo computador naquele momento.
- Grande parte da memória primária é volátil, ou seja, se o computador perder energia a informação perde-se. Uma das excepções é a ROM que vamos ver mais para a frente.
- É muito mais rápida do que a memória secundária mas muito mais lenta que a memória de cache.
- É o sistema que serve de intermediário entre a memória não volátil (memória secundária) e o processador.
- Nenhum computador consegue funcionar sem a memória primária.

Memória de um computador

Memória secundária

- Normalmente é o dispositivo de memória que, por ser não volátil, é usado para armazenar a informação mesmo com o computador desligado.
- É muito barata, o que permite que um computador tenha uma grande capacidade de armazenamento desta memória.
- No entanto, mesmo com os atuais SSD’s, é consideravelmente mais lenta do que a memória primária ou cache.
- Um processador, normalmente, não acede diretamente à informação que está neste tipo de memória.
- É a parte final do processamento efetuado pelo computador e é onde toda a informação que têm no computador reside.

Memória de um computador



Memória Primária

ROM e RAM

- Existem 2 grandes grupos de memória primária: ROM e RAM.
- ROM significa Read-Only-Memory ou seja, memória apenas para leitura.
- O conteúdo desta memória é introduzido pelo fabricante de um sistema e fica nesse estado até ao final da vida do sistema.
- Uma utilização muito conhecida deste tipo de memória é para guardar o firmware de um sistema.

Memória Primária

ROM e RAM

- RAM significa Random-Access Memory ou seja, memória de acesso aleatório. Este processo aleatório é, como vamos ver, um dos elementos fundamentais da computação.
- É, de longe, a forma de memória primária mais utilizada pois permite o acesso de forma extremamente rápida e eficiente à informação que nela está armazenada.
- É volátil, ou seja, assim que o sistema perder a energia o seu conteúdo desaparece.
- A memória RAM é uma base de memória temporária onde o computador armazena informação que ou está a necessitar ou necessita frequentemente para o processamento que está a realizar.
- A sua performance tem um enorme impacto na performance de um computador. Todos os computadores, sejam eles portáteis, telemóveis, tablets, etc, têm RAM.

RAM

Como funciona?

- Tal como o nome indica, o acesso a esta memória é feito de forma não sequencial, mas o que isso quer dizer?
- Significa que esta memória dá a possibilidade ao processador de aceder a qualquer posição da memória a qualquer altura. Para isso, o processador apenas precisa de saber o endereço de memória onde está a informação que pretende processar.
- Isto significa que o acesso, quer para escrita, quer para leitura é muito mais rápido do que a memória sequencial.
- (Outros tipos de memória, incluindo ROM, também permitem o acesso aleatório da informação)

RAM

Mas como?

- A memória RAM é composta por pares de transistores e condensadores. Cada par de transistor e condensador é uma célula de memória. Ou seja um bit de informação. Esse bit de informação é transformado em 0 ou 1 pelo processador.

- O condensador é um pequeno “balde” que recebe eletricidade. Se o “balde” estiver cheio com eletrões significa que a célula de memória está ativa (ou seja 1). Se o “balde” estiver vazio, a célula de memória está desativa (ou seja a 0).

- O transistor é a porta que decide ou não, consoante o que o processador decidir, se o condensador fica cheio ou vazio de eletrões.

RAM

0? 1? Uh?

- Se um bit, que é uma célula de memória, só pode ser 0 ou 1, isso significa que tudo o que é armazenado na RAM (e em quase todas as formas de memória) é em código binário. Mas como é que isso funciona mesmo?
- Toda a informação pode, em última análise, ser convertida para uma sequência (finita) de zeros e uns. Por exemplo, se eu quiser dizer os números, 1, 2, 3, 4, 5 e 6 posso dizer:

  0001
  0010
  0011
  0100
  0101
  0110

- Numa sequência de código binário, sempre que um dígito está ativo (com o seu valor a 1), o valor correspondente a esse dígito irá ser adicionado para calcular o valor total. E cada dígito vale 2^n, em que n é a posição do dígito na sequência.

Posição na sequência	5	4	3	2	1	0
Sequência	0	0	0	0	0	0
Valor quando ativo (a 1)	2^5	2^4	2^3	2^2	2^1	2^0

RAM

0? 1? Uh?

- Por exemplo, se quisermos escrever o número 35 em código binário:

Sequência	1	0	0	0	1	1
Valor quando ativo (a 1)	32				2	1

- Ou o número 41:

Sequência	1	0	1	0	0	1
Valor quando ativo (a 1)	32		8			1

- Ou o número 27:

Sequência	0	1	1	0	1	1
Valor quando ativo (a 1)		16	8		2	1

RAM

0? 1? Uh?

- Quando tivermos uma sequência de 0’s e 1’s que representam um número, podemos depois converter esse número para tudo o que for necessário.

  Palavras, pixels, instruções, notas de música, etc.

- Por exemplo, para palavras. Usando um codificador binário (um muito conhecido é o UTF-8) podemos converter a sequência binária num caracter.

- Por exemplo, se quisermos escrever a letra T em código binário, usando o codificador UTF-8, obtemos a sequência binária: 01010100 (ou seja, o número 84)

- Por exemplo, se quisermos escrever a frase “O Tomás já dorme!” Em código binário usando o codificador UTF-8, obtemos a sequência binária:

  01001111 00100000 01010100 01101111 01101101 11000011 10100001 01110011 00100000 01101010 11000011 10100001 00100000 01100100 01101111 01110010 01101101 01100101 00100001

RAM

0? 1? Uh?

- E como é que o processador sabe que uma dada sequência binária é um número, texto, pixel, etc?

  De forma muito simplificada, porque depende sempre do contexto onde essa sequência binária é chamada.

  Por exemplo, se estivermos a usar um editor de texto com um codificador UTF-8, todo o sistema (processador, programa, sistema operativo) sabe que aquele código binário representa caracteres. Se estivermos a ver uma fotografia, todo o sistema sabe que aquele código binário representa pixels.

RAM

Voltando ao funcionamento da RAM

- Como tínhamos visto, a memória RAM funciona através de células de memória que são compostas por pares de transistores e condensadores. Cada célula representa 1 bit (que ou está ativo (a 1) ou desativo (a 0)).
- 8 células de 1 bit perfazem 8 bytes. 1 Megabyte perfaz 1 milhão de bytes. Ou seja ,1 Megabyte tem 8 milhões de bit’s.
- Conforme vamos aumentando a capacidade de memória de um sistema, aumenta consideravelmente o número de células de memória.

RAM

Voltando ao funcionamento da RAM

- Se colocarmos todas as células de memória numa grelha, conseguimos aceder a qualquer célula usando as suas coordenadas. Independentemente da quantidade de células de memória que o sistema tenha.

  Por exemplo, na tabela seguinte, se eu quiser aceder à célula que tem o número 10, posso dizer que quero aceder à célula Bc

	a	b	c	d
A	1	2	20	23
B	1	1	**10**	1
C	4	1	2	33
D	6	5	19	13

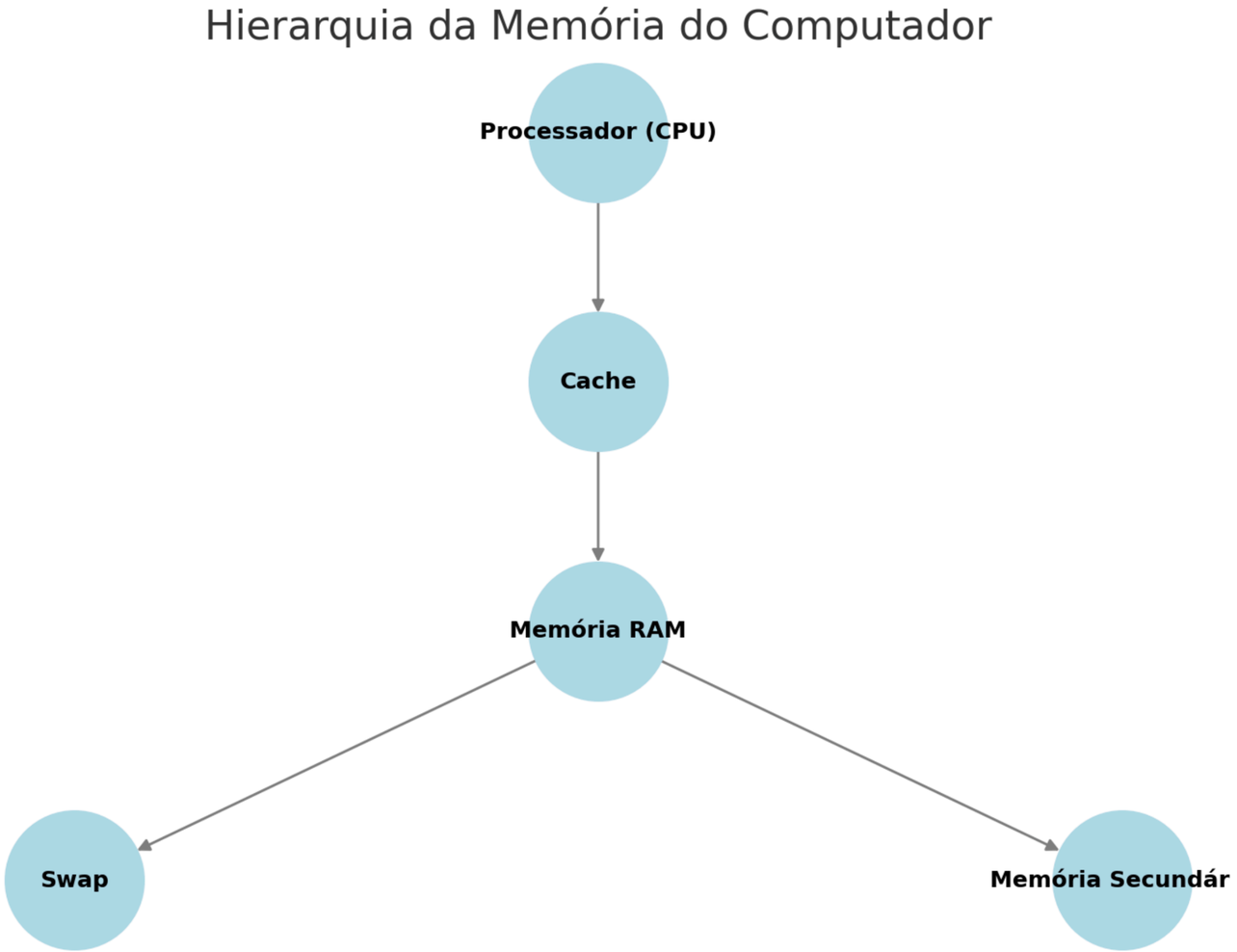
RAM

Voltando ao funcionamento da RAM

- De forma muito simplificada é assim que a informação está disposta na memória RAM. Através de células de memória que estão dispostas numa grelha.
- Assim, sempre que o processador necessitar de informação que esteja numa determinada célula pode aceder a essa informação usando as coordenadas da célula.
- A essas coordenadas chamamos de **endereço de memória**.
- Dentro dessa célula estará um condensador que estará ou não cheio com eletrões. Se estiver cheio o processador sabe que essa célula tem a informação 1 no código binário. Se estiver vazio, tem a informação 0.

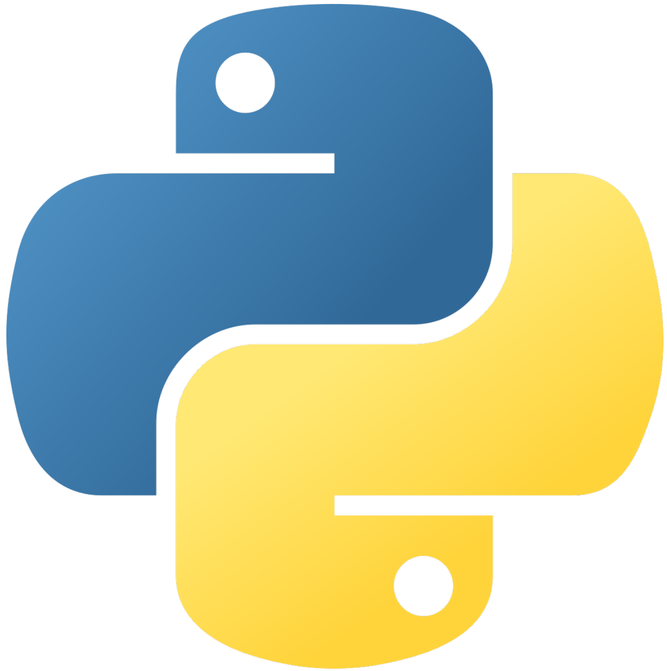
RAM

Diagrama de funcionamento da memória



Linguagens de Programação

Módulo 7 - Estruturas Dinâmicas

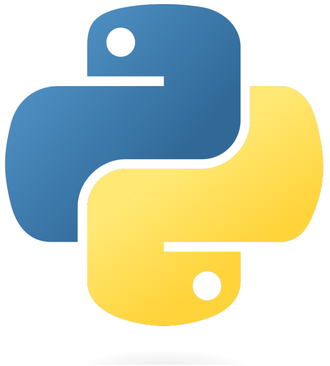


Gestão de Memória em Python

(matéria)

Python

Módulo 7 - Gestão de memória em Python



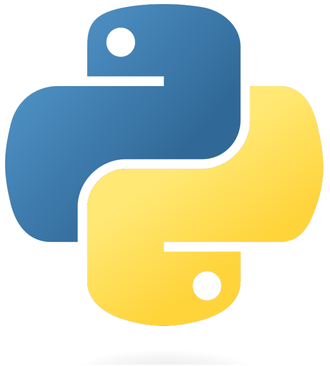
- O Python tem um sistema de **gestão automática de memória**, o que significa que os programadores não precisam (geralmente) de preocupar-se em alocar ou libertar memória manualmente.

- Para isso, Python utiliza um mecanismo de **Garbage Collection (GC)** baseado na **contagem de referências** e na deteção de **ciclos de referência**.

- Mas o que é isso de alocar memória? Ou libertar? E o que é o Garbage Collection? E contagem de referências? O que são referências?

Python

Módulo 7 - Gestão de memória em Python



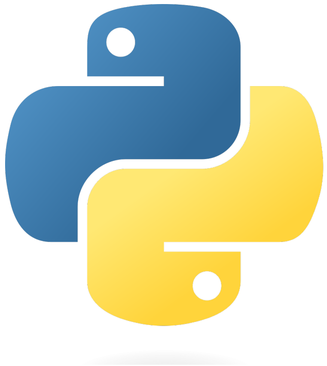
- **Alocar memória**, em computação, significa reservar um espaço na memória do computador para armazenar dados temporários durante a execução de um programa.

- Por outro lado, **libertar memória** significa devolver esse espaço para que possa ser reutilizado por outros processos.

- A forma como a memória é gerida depende da linguagem de programação e do sistema operativo. Python trata disso automaticamente, mas em linguagens como C e C++, o programador é responsável pela alocação e libertação da memória.

Python

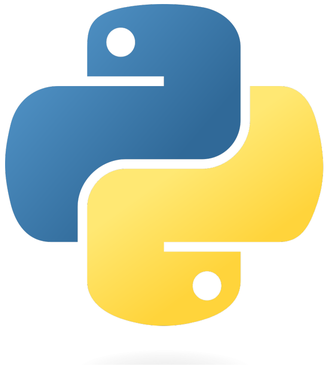
Módulo 7 - Gestão de memória em Python



- A memória pode ser alocada de duas formas: Alocação estática ou dinâmica.
- Alocação estática:
  - A memória é reservada antes da execução do programa.
  - O espaço alocado não pode ser alterado durante a execução (por exemplo um array).
    - Vantagens:
      - Mais rápida, pois a memória já está alocada antes do programa correr.
      - Evita fragmentação de memória.
    - Desvantagens:
      - Usa memória desnecessária se o espaço alocado não for totalmente utilizado.
      - Não pode ser redimensionada durante a execução.

Python

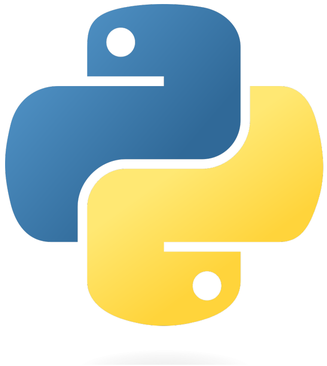
Módulo 7 - Gestão de memória em Python



- Alocação dinâmica:
  - A memória é reservada durante a execução do programa.
  - Permite alocar apenas a quantidade necessária no momento.
  - O programador pode decidir **quando** e **quanto** de memória alocar e libertar.
  - Vantagens:
    - Permite maior flexibilidade, pois podemos alocar e redimensionar a memória conforme necessário.
    - Usa apenas a memória estritamente necessária, evitando desperdício.
  - Desvantagens:
    - Pode ser mais lenta, pois envolve operações adicionais do sistema operativo.
    - Pode causar **memory leaks** se o programador não libertar a memória corretamente (problema em linguagens como C, mas não em Python).

Python

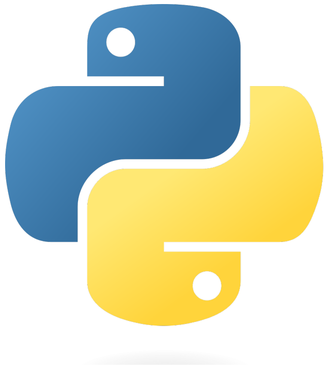
Módulo 7 - Gestão de memória em Python - Garbage Collection



- O **Garbage Collection (GC)** (ou Coletor de Lixo) é um mecanismo que gere automaticamente a **alocação** e **libertação** de memória em um programa. Ele é responsável por:
  - **Identificar** objetos que não estão mais a ser usados.
  - **Recolher e remover** esses objetos da memória RAM.
  - **Evitar** o desperdício de memória (memory leaks).

Python

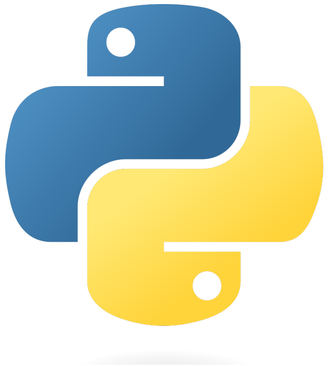
Módulo 7 - Gestão de memória em Python - Garbage Collection



- Mas como funciona?
  - O GC funciona em 3 etapas:
    - Alocação de memória
    - Contagem de referências
    - Cycle detection

Python

Módulo 7 - Gestão de memória em Python - Garbage Collection



- Alocação de memória já vimos. É a ação de reservar memória para algo.
- Mas o que são referências?
  - Uma **referência** é um **"endereço" ou "ponteiro"** que indica onde um objeto está armazenado na memória.
  - Quando criamos uma variável em Python, essa variável **não contém diretamente o valor**, mas sim **aponta para um local na memória** onde o valor está armazenado.

Python

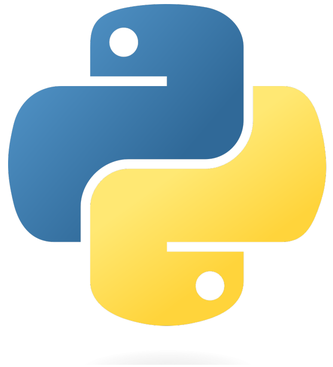
Módulo 7 - Gestão de memória em Python - Garbage Collection

```
x = 10

# A variável x não tem o valor 10 mas sim o endereço de memória onde o valor 10 está guardado
# O x é uma referência para o espaço de memória onde o valor 10 está guardado
```

- Em muitas linguagens de programação, chamamos a isto um ponteiro. E tem que ser criado manualmente. Ou seja, criamos a variável com o valor e depois uma segunda variável que vai apontar para o endereço de memória da primeira.
- Em Python isso acontece logo ao criarmos uma variável.

Python

Módulo 7 - Gestão de memória em Python - Garbage Collection



- No entanto, nem todas as variáveis são criadas da mesma forma. Experimenta o seguinte código:

```
x = 10
y = x

y = 20

print(y)
print(x)
```

- O que acontece?

- Agora experimenta o seguinte código:

```
x = [1, 2, 3, 4, 5]
y = x

y.append(6)

print(x)
print(y)
```

Python

Módulo 7 - Gestão de memória em Python - Garbage Collection

- Nos exemplos anteriores, conseguimos ver que quando criamos um x que tem apenas um número, equiparamos um y ao x e alteramos o y, o x mantém-se igual.

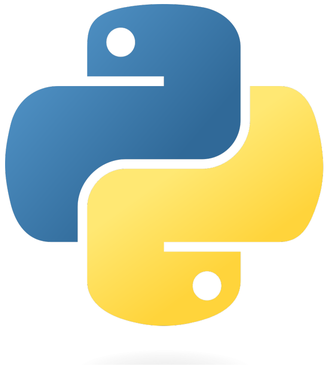
- No entanto, se criarmos uma lista e depois copiarmos essa lista para outra variável e alterarmos essa última lista, ambas as listas são alteradas.

```
x = 10
y = x

y = 20

print(y) # 20
print(x) # 10
```

```
x = [1, 2, 3, 4, 5]
y = x

y.append(6)

print(x) # [1, 2, 3, 4, 5, 6]
print(y) # [1, 2, 3, 4, 5, 6]
```

Python

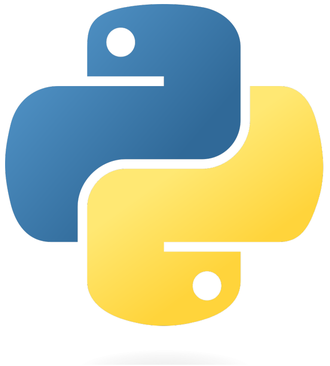
Módulo 7 - Gestão de memória em Python - Garbage Collection



- Em todas as situações, ao criar o x estamos a criar referência para valores, ou seja, estamos a guardar o local onde o valor da variável está armazenado.
- Quando criamos o y e colocamos lá o conteúdo do x, estamos a dizer que o y aponta para onde o x aponta, ou seja, para o valor do x.
- No entanto, quando alteramos o primeiro y, o x mantém-se igual. Isto é porque alguns tipos de dados (como o int) são o que se chamam de dados imutáveis.
- Já quando aplicamos o mesmo processo a uma lista, ambas as listas são alteradas, isto é porque as listas são dados mutáveis.

Python

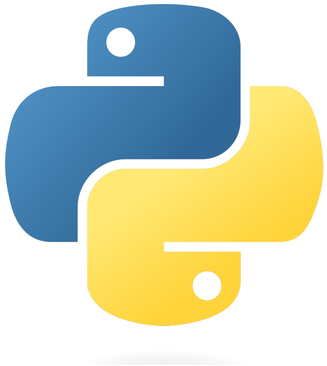
Módulo 7 - Gestão de memória em Python - Garbage Collection



- Tipos de dados imutáveis são aqueles que **não podem ser modificados** depois de serem criados.
- Se tentarmos alterar um valor imutável, Python cria **um novo objeto** na memória em vez de modificar o original.
- Ou seja, o y é criado como uma referência para o valor de x mas, quando o alteramos, é criado um novo objeto com o novo valor para onde y vai apontar. No entanto o x mantém-se a apontar para onde já apontava.
- Principais Tipos Imutáveis em Python:
  - Números: `int`, `float`
  - Strings: `str`
  - Tuplas: `tuple`
  - Booleans: `bool`
  - Bytes: `bytes`

Python

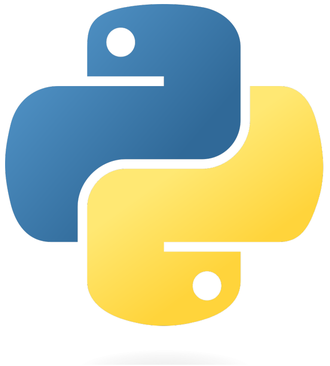
Módulo 7 - Gestão de memória em Python - Garbage Collection



- Já os tipos de dados mutáveis são aqueles que podem ser modificados após a sua criação.
- Se alterarmos um objeto mutável, ele continua no **mesmo local de memória**, e qualquer referência a ele verá a alteração.
- Ou seja, quando criamos a lista x, alocamos o espaço na memória para a lista e colocamos no x uma referência para esse espaço. Depois quando equiparamos o y, este fica também a apontar para o mesmo espaço de memória. Quando alteramos o y (ou o x), o objeto continua a ser o mesmo e ambas as referências continua a apontar para ele.
- Principais Tipos Mutáveis em Python:
  - Listas
  - Dicionários
  - Conjuntos
  - Objetos personalizados (classes definidas pelo programador)

Python

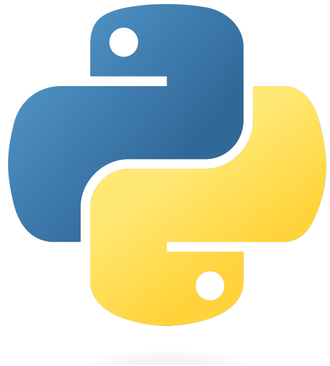
Módulo 7 - Gestão de memória em Python - Garbage Collection



- Vamos então regressar ao funcionamento do Garbage Collection.
- Tínhamos visto que o primeiro passo é a alocação de memória que, no Python, é feito de forma automática.
- A seguir temos a contagem de referências.
- E nesta fase é feito precisamente isso. São contadas referências para variáveis.

Python

Módulo 7 - Gestão de memória em Python - Garbage Collection

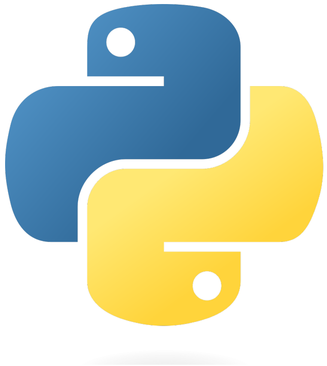


- De forma muito simples, o GC mantém um registo de quantas referências estão associadas a um valor.

```
x = 10 # É criada a primeira referência para o objeto 10
y = x # É criada uma segunda referência para o objeto 10

y = 20 # A referência y é alterada para o objeto 20
# ou seja, a referência y deixa de apontar para o objeto 10 e passa a apontar para o objeto 20

# Neste momento há uma referência para o objeto 10 e outra para o objeto 20
# Se apagarmos a referência x, o objeto 10 deixa de ter referências para ele
# Logo o garbage collector irá apagar o objeto 10 da memória
```

Python

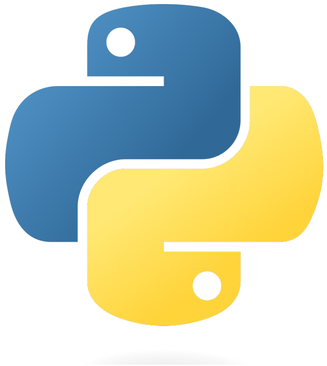
Módulo 7 - Gestão de memória em Python - Garbage Collection



- Ou seja, se um objeto ainda tiver 1 ou mais referências para ele, ele é mantido.
- Se tiver 0 referências, ele é chamado de órfão e é apagado da memória.
- Todo este processo é feito de forma automática em Python através do G.C.
- Depois temos a terceira etapa do G.C. que é a coleta de ciclos de referência.
- Esta etapa é uma etapa de redundância para garantir que nenhuma variável que deva ser apagada fique por apagar (Memory leak)

Python

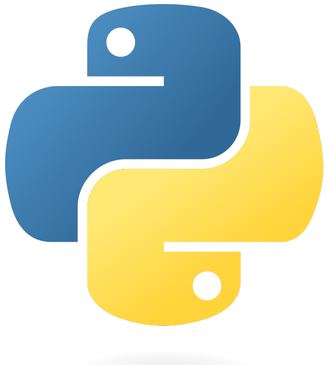
Módulo 7 - Gestão de memória em Python - Garbage Collection



- A contagem de referências pode ter um problema:
- Se um objeto referenciar a si mesmo ou dois objetos se referirem mutuamente, o **contador de referências nunca chega a zero**. Python resolve isso com a **deteção de ciclos**.
- Para evitar que isto aconteça, o G.C. percorre todos os objetos em memória e verifica todas as referências que só são acessadas mutuamente.
- Se encontrar duas referências que apontem uma para a outra, remove o objeto associado a essas referências.

Python

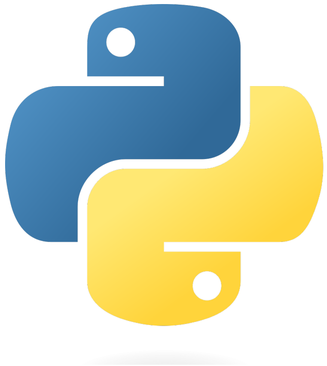
Módulo 7 - Gestão de memória em Python - Garbage Collection



- O motor de Python está sempre a aplicar estes dois processos (contagem de referências e coleta de ciclos de referência) em busca de objetos órfãos de referências ou referências circulares.
- Ou seja:
  - Se um objeto tiver zero referências apontadas a ele, é removido
  - Se houver um ciclo de referências, este é removido assim como qualquer objeto que possa estar a ele associado.

Python

Módulo 7 - Gestão de memória em Python - Resumindo



- Dados imutáveis
  - Consideremos o seguinte código executado **FORA** de uma função:

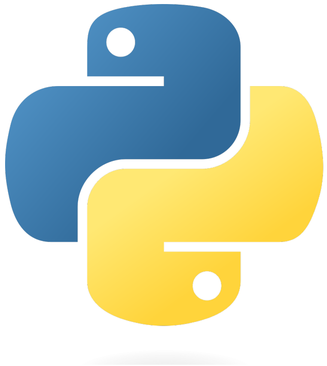
```
x = 10
```

  - O Python vai primeiro verificar a sintaxe e depois executa o código.
  - Ao executar o código cria um objeto do tipo inteiro.
  - Uma vez que Python trata todos os valores como objetos, quando escrevemos x = 10 vai:
    - Verificar se o valor 10 já está em memória.
    - Se não existir, vai criar um novo objeto int com o valor 10 no Heap
    - O objeto 10 recebe um ID único que é o seu endereço de memória

```
x = 10

print(id(x)) # 140732674000368
```

Python

Módulo 7 - Gestão de memória em Python - Resumindo

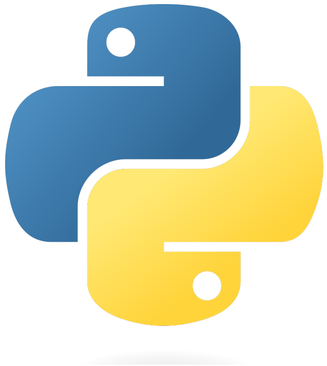


- Variáveis São Apenas Referências
  - A variável `x` não armazena diretamente o valor `10`.
  - `x` é apenas um **ponteiro** (referência) que aponta para o endereço do objeto `10`.
- No slide anterior foi dito que quando criamos o x = 10, o Python verifica se o 10 já está em memória. Isso significa que se tivermos o seguinte código, todas as variáveis (referências) apontam para o mesmo 10?

```
x = 10
y = 10

# Quer o x, quer o y são referências para o mesmo objeto
# Ou seja, quando criamos o y, o Python vai verificar se já existe um objeto com o valor 10 e se existir,
# ele vai criar uma nova referência para esse objeto.
```

Python

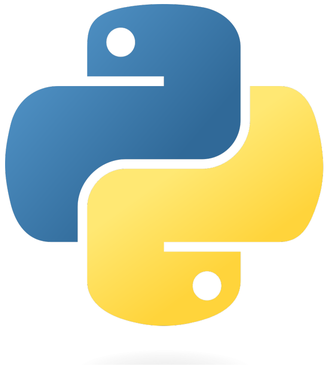
Módulo 7 - Gestão de memória em Python - Resumindo



- A isto chamamos de Integer Pooling para números pequenos (de -5 a 256).
- Neste caso, se criarmos variáveis com o mesmo valor, o Python não cria objetos novos, mas sim faz com que todas as referências apontem para o mesmo objeto.
- Isto economiza memória e melhora o desempenho.

Python

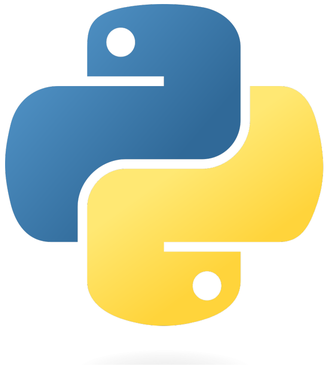
Módulo 7 - Gestão de memória em Python - S.O.



- E o Sistema Operativo? O que faz com tudo isto?
- Agora que Python criou o objeto, o **sistema operativo** precisa garantir que ele **tem um local na RAM**. O processo, de forma muito simples, acontece assim:
  - Python pede ao sistema operativo um espaço na RAM para armazenar o objeto `10` (caso ainda não exista).
  - O sistema operativo usa um mecanismo chamado "Heap Allocation" para alocar espaço na RAM.
  - O endereço desse espaço na memória é devolvido para o Python, que agora mantém essa referência.
- O sistema operativo apenas vê que o Python está a pedir memória e responde com um **endereço válido**.

Python

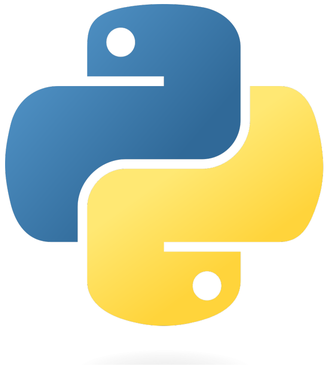
Módulo 7 - Gestão de memória em Python



- E se alterarmos o valor de uma variável imutável?
- Neste caso o Python não altera o valor na memória. Em vez disso:
  - Python cria um novo objeto com o novo valor no Heap.
  - A variável alterada aponta para o novo objeto.
  - O antigo objeto não é mais referenciado pela variável
  - Se o objeto não tiver mais referências é apagado pelo G.C.

Python

Módulo 7 - Gestão de memória em Python - Resumindo



- Dados mutáveis
  - Consideremos o seguinte código:

```
x = [1, 2, 3]
```

  - Quando esta linha é executada, o Python precisa de:
    - Criar um objeto mutável (lista) na memória
    - Guardar os elementos da lista separadamente na memória
    - Criar uma referência para a variável x que aponte para o objeto.

Python

Módulo 7 - Gestão de memória em Python

- Então, quando o código é executado:
  - O interpretador de Python solicita o espaço de memória no Heap para armazenar a estrutura da lista
  - Os elementos da lista são armazenados separadamente também no Heap.
  - A variável x não armazena os valores diretamente, mas sim o endereço da estrutura da lista.
- Experimenta o seguinte código:

```
x = [1, 2, 3]
print(id(x))  # ID da lista na memória Heap
print(id(x[0]), id(x[1]), id(x[2]))  # IDs dos elementos individuais
```

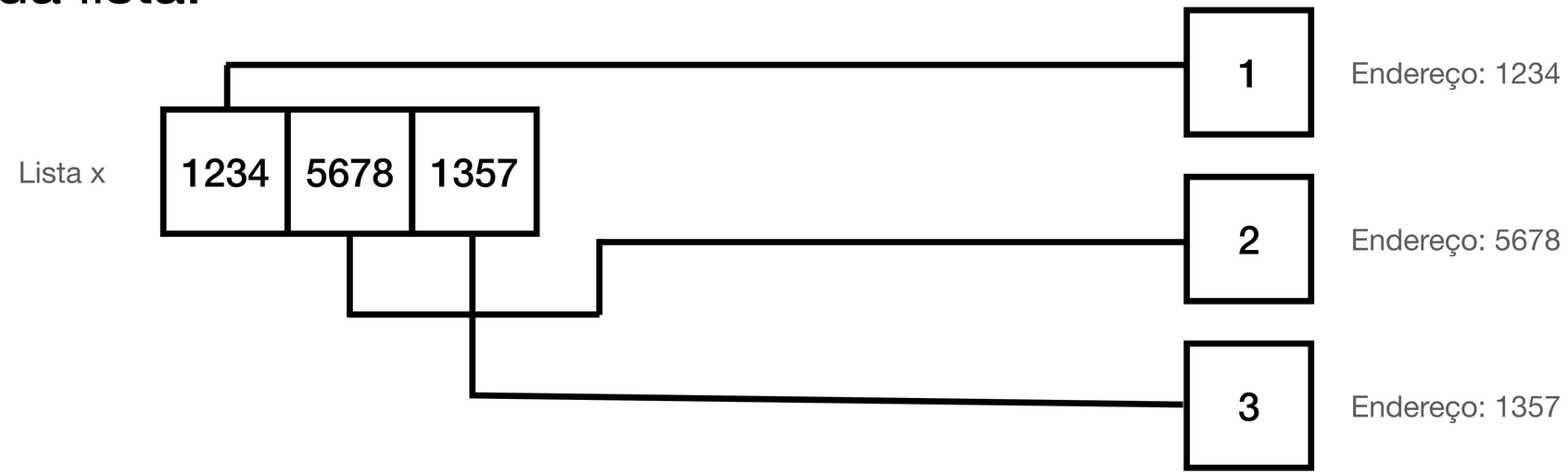
Python

Módulo 7 - Gestão de memória em Python

- Então, se x contém o endereço (referência) para a estrutura lista que está armazenada no Heap, e se os elementos da lista estão armazenados também separadamente no Heap, o que contém, na realidade, a estrutura lista? E como é que o Python sabe onde estão todos os elementos da lista?

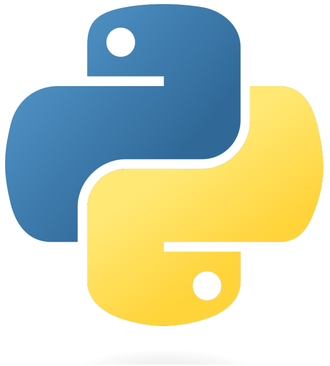
- A estrutura lista contêm, em cada elemento, uma referência para cada um dos valores da lista.

```
x = [1, 2, 3]
```



Python

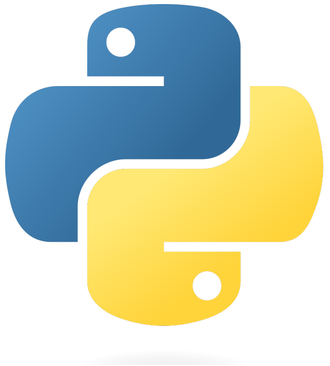
Módulo 7 - Gestão de memória em Python - S.O.



- Quando Python cria um objeto mutável:
  - Pede ao S.O. um bloco de memória na RAM
  - O Sistema Operativo reserva espaço na Heap e devolve o seu endereço de memória
  - Python mantém um ponteiro para esse endereço de memória

Python

Módulo 7 - Gestão de memória em Python



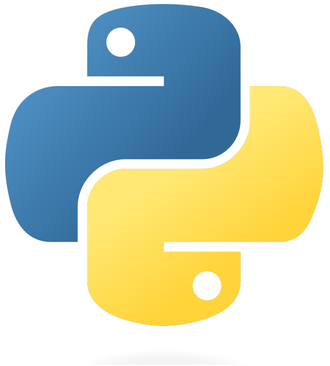
- E se alterarmos um objeto mutável?
- As listas podem ser alteradas, os dicionários também. O que acontece aí?

```
x = [1, 2, 3]
x.append(4)  # Adicionamos um novo elemento à mesma lista na Heap
print(x)  # [1, 2, 3, 4]
```

- Neste caso, é reservado na Heap o espaço para alocar o novo elemento e é adicionada uma referência para ele na estrutura da lista.

Python

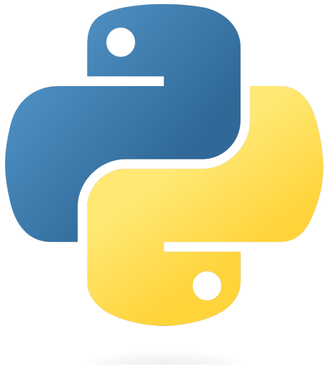
Módulo 7 - Gestão de memória em Python - Objeto? Heap? Stack?



- Temos falado sobre objetos e memória Heap, mas o que é isso em concreto?
- E já agora, temos falado que se tivermos, por exemplo, x = 10, o x é uma referência para o dez, mas onde fica guardada essa referência?

Python

Módulo 7 - Gestão de memória em Python - Objeto? Heap? Stack?

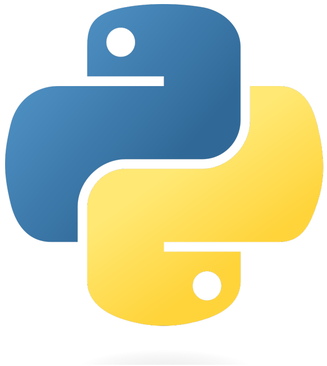


- Um objeto é uma estrutura de dados que combina valores (dados) e comportamentos (métodos/funções).
- Em Python, tudo é um objeto.

```
x = 10  # '10' é um objeto do tipo int
y = "Olá"  # "Olá" é um objeto do tipo str
```

Python

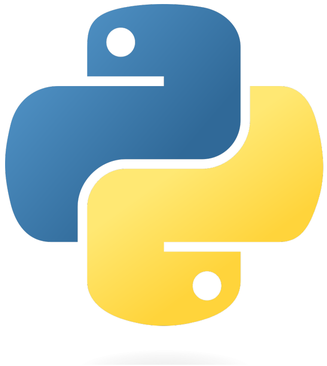
Módulo 7 - Gestão de memória em Python - Objeto? Heap? Stack?



- Cada objeto em Python tem:
  - **Um tipo** (`type(obj)`) → Define o que ele é (exemplo: `int`, `str`, `list`).
  - **Um ID (endereço na memória)** (`id(obj)`) → Onde ele está armazenado na RAM.
  - **Atributos e Métodos** (`dir(obj)`) → Funcionalidades associadas ao objeto.

Python

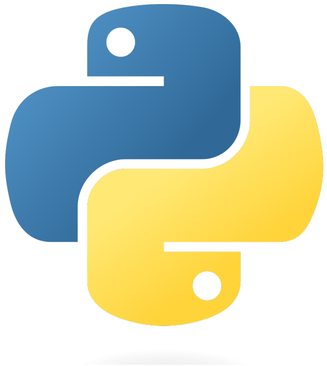
Módulo 7 - Gestão de memória em Python - Objeto? Heap? Stack?



- E o Heap e o Stack?
- Quando um programa é executado, a **memória RAM** é dividida em diferentes regiões, sendo as duas mais importantes:
  - **Stack (Pilha)** – Usada para armazenar, referências, valores das variáveis locais (em determinadas situações) e chamadas de função.
  - **Heap** – Usada para armazenar valores das variáveis, objetos dinâmicos e estruturas de dados grandes.

Python

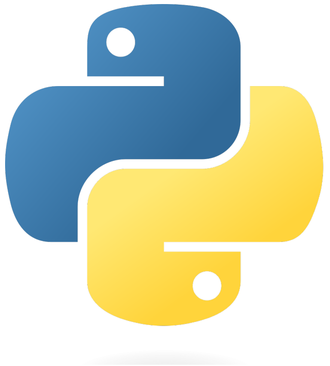
Módulo 7 - Gestão de memória em Python - Stack



- A **Stack** segue uma estrutura LIFO (Last In, First Out)
- Na Stack, como já vimos, vão ser armazenadas referências, variáveis locais, chamadas de funções. Mas o que é que isso quer dizer?
  - Sempre que uma função é executada, um bloco de memória é adicionado ao topo da Stack, esse bloco chama-se Stack Frame.
  - Nesse bloco de memória vão estar as referências para as variáveis locais da função (incluindo os seus parâmetros), caso seja possível (ou seja, valores pequenos de -5 até 256) vão também estar as variáveis locais e qualquer chamada de função que seja efetuada na função mãe.

Python

Módulo 7 - Gestão de memória em Python - Stack



- Quando a função termina, o Stack Frame é removido da Stack e todos os seus dados são removidos automaticamente.

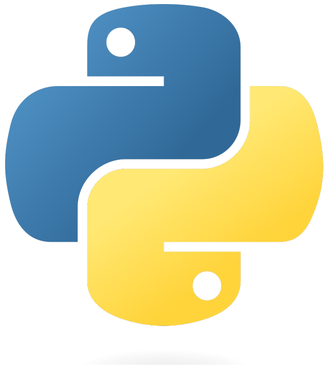
- Isso significa que todos os objetos que estejam no Heap e estivessem a ser referenciados no Stack Frame que foi apagado, podem ficar órfãos. E nesse caso, serão removidos pelo G.C.

- A Stack tem um tamanho limitado. E em algumas situações podemos esgotar esse espaço. A isto chamamos StackOverflow.

- Um exemplo dessa ocorrência é quando usamos recursividade sem condição de paragem. Por cada chamada da função recursiva, será criado um Stack Frame. Eventualmente todo o espaço irá ser usado.

Python

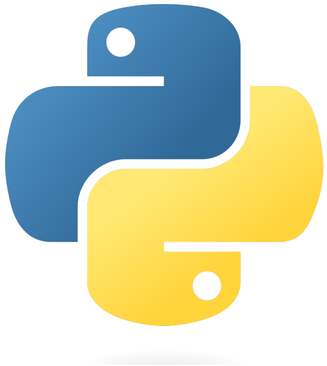
Módulo 7 - Gestão de memória em Python - Heap



- E o Heap?
- Tal como a Stack, o Heap também é uma zona da memória RAM.
- Mas é diferente da **Stack**, onde os dados são armazenados de forma temporária e organizada, a Heap é **gerida dinamicamente** pelo programa e pode crescer conforme necessário.

Python

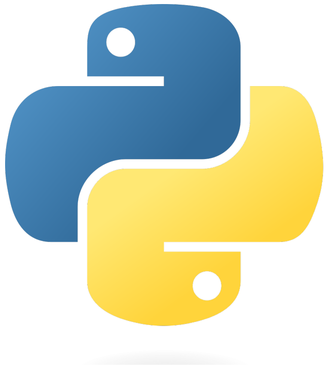
Módulo 7 - Gestão de memória em Python - Heap



- Num sistema operativo, a Heap é concebida para armazenar grandes quantidades de informação que precisam persistir independentemente da execução de funções.
- Não segue uma ordem especifica (ao contrário da Stack). Os seus dados podem ser acedidos a qualquer momento.
- É gerida pelo programador ou pelo G.C.. Ao contrário da Stack onde os dados são adicionados / removidos automaticamente com o inicio / fim de funções.
- Tem um tamanho flexível.
- Todos os objetos mutáveis são armazenados na Heap. No entanto, as suas referências são armazenadas na Stack.

Python

Módulo 7 - Gestão de memória em Python - Heap & Stack



```python
x = 10  # Variável global, imutável e pequena (entre -5 e 256)
# O número 10 está no Integer Pool e pode ser armazenado diretamente na Stack.
# A variável x está na Stack e armazena a referência para o número 10.

v = 1000  # Variável global, imutável e grande (fora do intervalo -5 e 256)
# Como 1000 está fora do intervalo do Integer Pool, ele é armazenado na Heap.
# A referência para esse valor é armazenada na Stack, mas o próprio valor está na Heap.

y = [1, 2, 3]  # Objeto mutável (lista)
# A variável y está na Stack e contém uma referência para a estrutura da lista, que está na Heap.
# A estrutura da lista contém referências para os elementos da lista.
# Os elementos da lista podem estar:
#   - No Integer Pool (caso sejam pequenos números inteiros, como 1, 2 e 3).
#   - Na Heap (caso sejam números grandes ou outros objetos mutáveis).
# Assim, a estrutura da lista contém ponteiros que apontam para os endereços dos valores dos elementos.
```

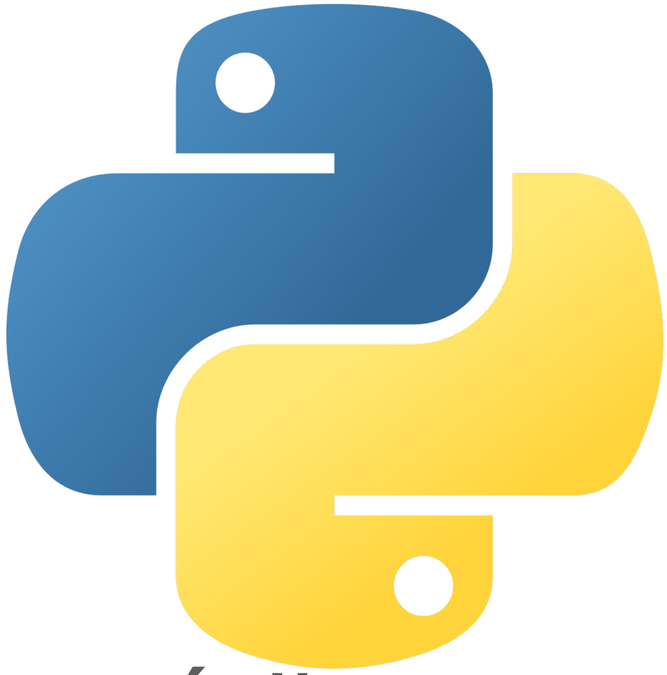
Python

Módulo 7 - Gestão de memória em Python - Heap & Stack

```python
def funcao(z):
    w = 20  # Variável local imutável pequena (entre -5 e 256)
    # O número 20 está no Integer Pool e pode ser armazenado diretamente na Stack.
    # A variável w está na Stack e armazena uma referência para o número 20.

    u = 1000  # Variável local imutável grande (fora do intervalo -5 a 256)
    # O número 1000 já existe na Heap (reutilizado da variável global 'v' caso ainda esteja referenciado).
    # A variável u está na Stack e contém uma referência para esse número na Heap.

funcao(10)
# Quando a função é chamada, um novo contexto de execução (Stack Frame) é criado na Stack.
# Dentro desse Stack Frame:
#   - A variável z é armazenada na Stack e aponta para o número 10 (que está no Integer Pool).
#   - A variável w é armazenada na Stack e aponta para o número 20 (que também está no Integer Pool).
#   - A variável u é armazenada na Stack e contém uma referência para o número 1000 (que está na Heap).
# Quando a função termina, o Stack Frame é removido da Stack.
# Todas as variáveis locais (z, w, u) são removidas automaticamente.
# Objetos órfãos na Heap (sem referências na Stack) podem ser removidos pelo Garbage Collector,
# mas essa remoção não acontece imediatamente — apenas quando Python decidir liberar memória.
```

Linguagens de Programação

Módulo 7 - Estruturas Dinâmicas

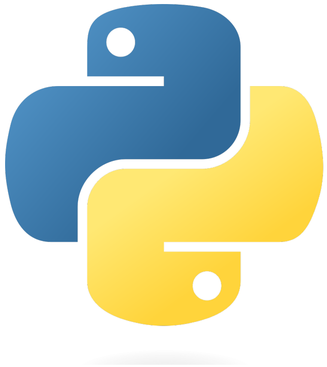


Compilação e Interpretação de código

(matéria)

Python

Módulo 7 - Estruturas Dinâmicas

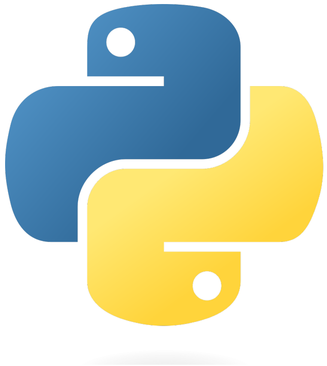


- O que acontece num computador quando executamos um bloco de código?
- Como é que as instruções de uma linguagem de programação são convertidas em instruções para o nosso computador?

- Cada linguagem tem o seu modo de conversão, neste módulo iremos aprofundar como o Python o faz.

Python

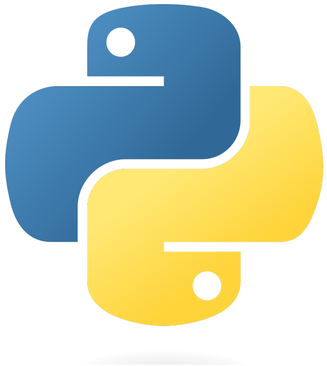
Módulo 7 - Estruturas Dinâmicas



- De forma muito simples:
  - Primeiro precisamos do código fonte, ou seja, o nosso código.
  - Depois o interpretador converte o código para uma linguagem intermédia que é agnóstica relativamente ao sistema operativo (ou seja, funciona em todos os S.O.)
  - O código dessa linguagem é depois introduzido numa máquina virtual especial.
  - Essa máquina virtual comunica depois, cada uma das instruções diretamente com o Sistema Operativo.
  - O Sistema Operativo recebe essas instruções e passa-as para o processador para serem processadas.

Python

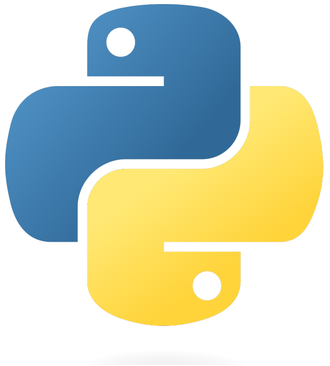
Módulo 7 - Estruturas Dinâmicas



- Mas… Máquina Virtual? Compilar? Interpretar? O que é isso?
- Uma **Máquina Virtual (VM)** é um software que emula um ambiente computacional, permitindo que um sistema operativo ou um programa seja executado independentemente do hardware físico.
- Em outras palavras, uma máquina virtual simula um computador dentro de outro computador.

Python

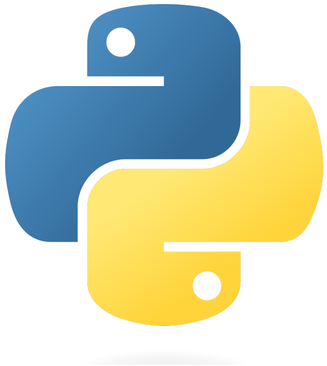
Módulo 7 - Estruturas Dinâmicas



- Vantagens das Máquinas Virtuais
  - ✅ **Portabilidade** – O mesmo código pode ser executado em diferentes sistemas.
  - ✅ **Segurança** – Mantém os processos isolados, evitando interferências no sistema principal.
  - ✅ **Flexibilidade** – Permite testar diferentes sistemas operativos e aplicações sem afetar o sistema real.
  - ✅ **Eficiência no Desenvolvimento** – Facilita testes e simulações de diferentes ambientes.
- Desvantagens
  - ❌ **Desempenho** – A execução dentro de uma VM pode ser mais lenta que no hardware real.
  - ❌ **Consumo de Recursos** – Pode exigir mais memória RAM e CPU para funcionar corretamente.

Python

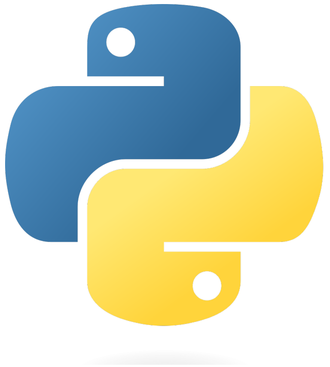
Módulo 7 - Estruturas Dinâmicas - Compilação



- Compilação é o processo de conversão do código-fonte de uma linguagem de programação de alto nível para código de máquina antes da execução.
- Esse processo é realizado por um **compilador**, que analisa e traduz todo o código de uma só vez, gerando um **executável** independente do código original.
- Linguagens compiladas: C, C++, Java, etc

Python

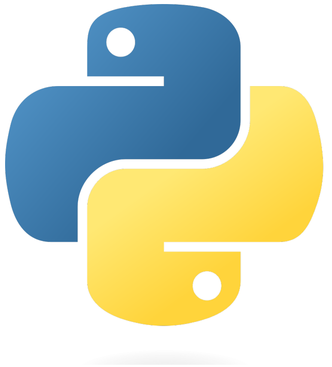
Módulo 7 - Estruturas Dinâmicas - Interpretação



- A **interpretação** é o processo onde um **interpretador** lê e executa o código-fonte linha a linha, sem gerar um executável independente.
- O código-fonte é traduzido dinamicamente durante a execução.
- Linguagens interpretadas: Python, JavaScript, PHP, etc.

Python

Módulo 7 - Estruturas Dinâmicas - Compilação vs Interpretação



Característica	Compilação	Interpretação
Execução	Código é traduzido antes da execução.	Código é traduzido durante a execução.
Velocidade	Mais rápido, pois já está em código de máquina.	Mais lento, pois traduz linha a linha.
Erro detectado	Durante a compilação.	Durante a execução.

Python

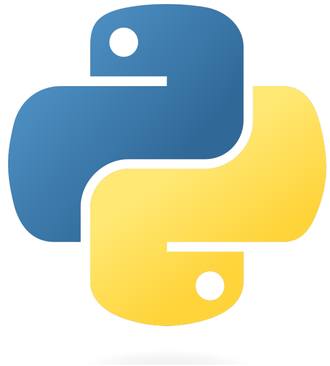
Módulo 7 - Estruturas Dinâmicas - Python

- Python é uma **linguagem interpretada**, mas com uma etapa intermediária de compilação para **bytecode**. Vamos detalhar passo a passo o que acontece quando executamos um script Python.
- **Código-fonte**
  - Quando um programador escreve um código Python, ele está a escrever um **código de alto nível** num ficheiro com extensão .py.
  - Esse código contém instruções que um humano consegue compreender facilmente, mas que um computador não consegue executar diretamente.

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

print(saudacao("Nuno"))
```

Python

Módulo 7 - Estruturas Dinâmicas - Python

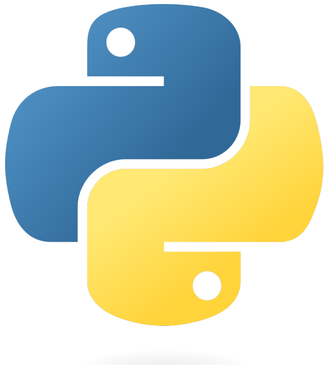


- **Compilação para ByteCode:**
  - Ao executar o código Python, a primeira coisa que o **interpretador Python** faz é convertê-lo para um formato chamado **bytecode**. Esta conversão ocorre de forma automática e transparente.
  - O **bytecode** é uma representação de baixo nível do código, mas ainda não é código de máquina.
  - O bytecode é independente do sistema operativo e do hardware, permitindo portabilidade entre diferentes plataformas.
- **O que acontece nesta etapa?**
  - Python lê o ficheiro .py e verifica se há erros de sintaxe.
  - Se não houver erros, o código é transformado em **bytecode** (.pyc).
  - O bytecode pode ser armazenado na pasta __pycache__, o que evita a recompilação em execuções futuras.
- O código do slide anterior converte-se para estas instruções em bytecode:

```
  0 RESUME                   0
  2 LOAD_CONST               1 ('Olá, ')
  4 LOAD_FAST                0 (nome)
  6 FORMAT_VALUE             0
  8 LOAD_CONST               2 ('!')
 10 BUILD_STRING             3
 12 RETURN_VALUE
```

Python

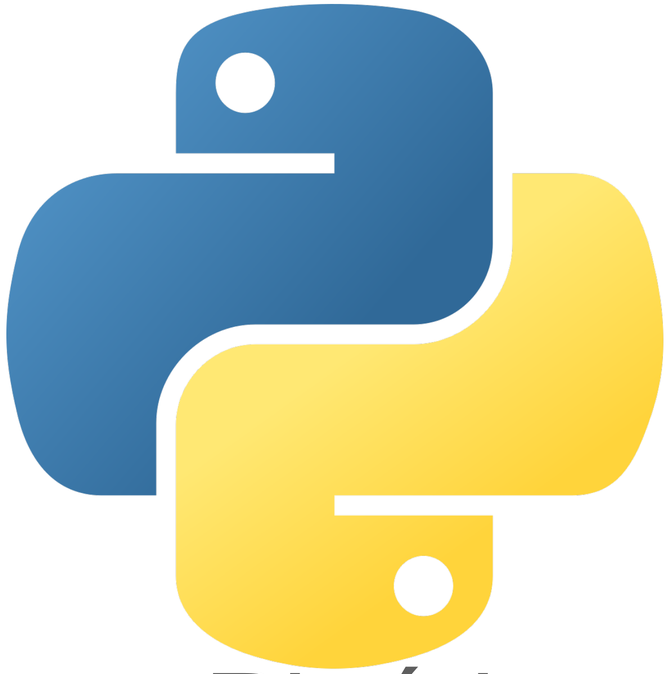
Módulo 7 - Estruturas Dinâmicas - Python



- **A Máquina Virtual Python (PVM)**
  - Agora que temos o bytecode, ele precisa de ser executado. É aqui que entra a **Máquina Virtual Python (PVM)**, que interpreta o bytecode e executa as instruções **linha a linha**.
  - A PVM age como um intermediário entre o código Python e o sistema operativo:
    - Ela lê as instruções do bytecode.
    - Traduz para chamadas apropriadas ao processador e ao sistema operativo.
    - Executa o programa.

Linguagens de Programação

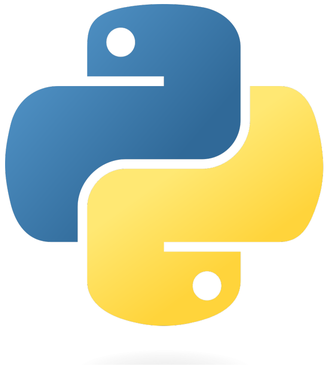
Módulo 7 - Estruturas Dinâmicas



Filas, Pilhas, Listas Ligadas, Arvores Binárias

Python

Módulo 7 - Estruturas Dinâmicas

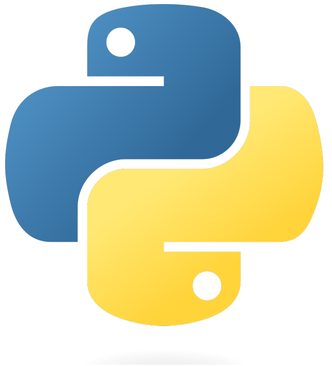


- As estruturas dinâmicas de dados são estruturas que podem crescer ou diminuir de tamanho durante a execução do programa, sem necessidade de definir previamente a sua capacidade.

- Diferente das **estruturas estáticas** (como arrays, que têm tamanho fixo), as estruturas dinâmicas permitem **adicionar, remover e reorganizar elementos conforme necessário**.

Python

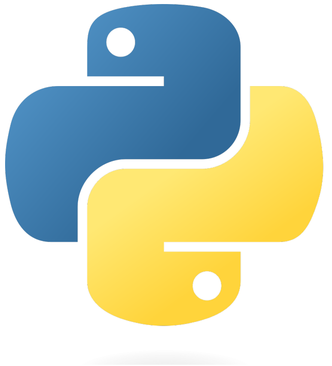
Módulo 7 - Estruturas Dinâmicas



- Características Principais:
  - ✅ **Alocação dinâmica de memória** → O espaço é alocado conforme necessário, otimizando o uso da memória.
  - ✅ **Flexibilidade** → Permitem adicionar e remover elementos facilmente, sem desperdício de espaço.
  - ✅ **Uso de Ponteiros/Referências** → Normalmente, os elementos estão ligados entre si através de **referências** (endereços de memória).
  - ✅ **Mais eficientes para inserções e remoções frequentes** → Comparadas a estruturas estáticas, onde estas operações podem ser mais custosas.

Python

Módulo 7 - Estruturas Dinâmicas



- Exemplos de Estruturas Dinâmicas
  - **Pilhas (Stacks)** → Estruturas baseadas no princípio **LIFO (Last In, First Out)**. Exemplo: Histórico de navegação num browser.
  - **Filas (Queues)** → Estruturas baseadas no princípio **FIFO (First In, First Out)**. Exemplo: Processamento de pedidos numa impressora.
  - **Listas Ligadas (Linked Lists)** → Cada elemento aponta para o próximo, permitindo crescimento dinâmico. Exemplo: Estruturas de menus interativos.
  - **Árvores (Trees)** → Estruturas hierárquicas, onde cada elemento pode ter filhos. Exemplo: Sistemas de ficheiros no computador.

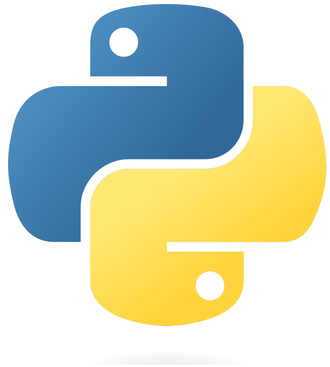
Python

Módulo 7 - Estruturas Dinâmicas

Característica	Estruturas Estáticas (ex.: Arrays)	Estruturas Dinâmicas
Tamanho	Fixo (definido na criação)	Variável (cresce/diminui conforme necessário)
Uso de memória	Pré-alocada (pode desperdiçar espaço)	Alocação dinâmica (uso eficiente da memória)
Inserção/Remoção	Pode ser demorada (realoca elementos)	Rápida, pois basta ajustar referências
Acesso a elementos	Acesso rápido (O(1)) por índice	Percorrer até encontrar (O(n) em listas ligadas)

Python

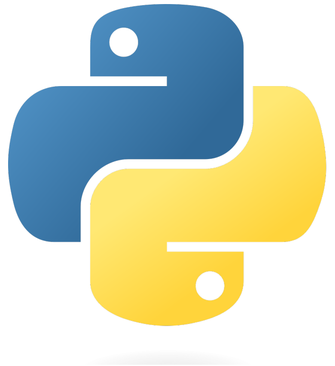
Módulo 7 - Estruturas Dinâmicas



- Então, mas se estruturas dinâmicas são estruturas que podem modificar o seu tamanho livremente, uma lista é uma estrutura dinâmica?
  - Depende da linguagem. Em Python **não é.**
  - Em Python uma lista é, na realidade um array de tamanho fixo.
  - Quando criamos uma lista em Python, é reservado um espaço fixo um pouco maior do que o necessário.
  - No entanto, se a lista crescer mais do que o esperado, é criado um novo array, é reservado novo espaço e a lista é copiada para esse novo espaço.
  - Ou seja, embora uma lista pareça uma estrutura dinâmica, no seu funcionamento não é.

Python

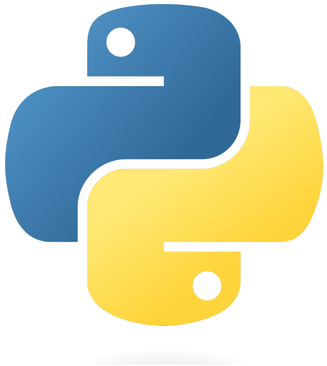
Módulo 7 - Estruturas Dinâmicas



- Antes de avançarmos para as 4 estruturas dinâmicas, precisamos de ver alguns conceitos chave. Classes, objetos, complexidade e nós.

Python

Módulo 7 - Estruturas Dinâmicas - Classes e objetos



- O que é uma Classe?
  - Uma **classe** é um **modelo (ou molde)** para criar objetos.
  - Ela define as características (atributos) e os comportamentos (métodos/funções) que os objetos terão.
- 📌 Exemplo no mundo real:
  - Imagina que estás a desenhar **um modelo para fabricar carros**.
  - O modelo define **características** como cor, marca e número de rodas.
  - Também define **comportamentos** como acelerar e travar.
  - Mas o modelo, por si só, **não é um carro** – ele apenas diz como um carro deve ser.

Python

Módulo 7 - Estruturas Dinâmicas - Classes e objetos

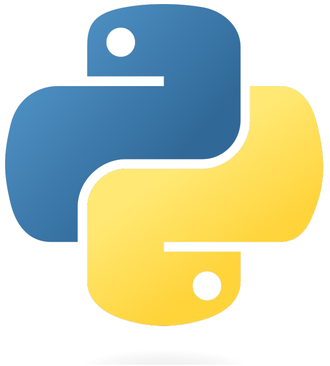
- Uma classe é composta por 3 partes:
  - Atributos - Os atributos são variáveis associadas a uma classe. Definem as características do modelo que é a classe.
    - Por exemplo, se tivermos uma classe Pessoa, os seus atributos seriam características como nome, idade, nacionalidade, altura, peso, etc.

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

- Neste exemplo, criamos uma classe pessoa com alguns atributos.

Python

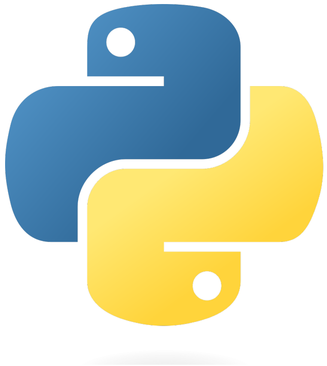
Módulo 7 - Estruturas Dinâmicas - Classes e objetos



- Uma classe é composta por 3 partes:
  - Propriedades: As **propriedades** são atributos especiais que permitem **controlar o acesso e modificação** de valores nos objetos.
  - Há várias propriedades que podemos aplicar a um atributo, mas normalmente temos duas que são usadas frequentemente:
    - GET -> A propriedade get permite ir buscar o valor de um atributo. Permite esconder o atributo de acessos indevidos ou permite processar/tratar os dados quando são obtidos.
    - SET -> A propriedade set permite definir um atributo. Permite criar condições para a definição de um atributo.

Python

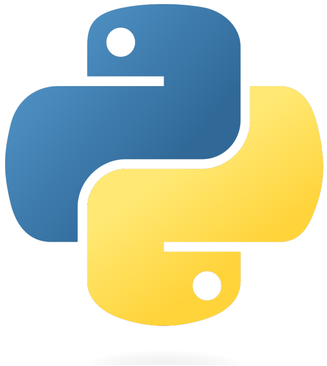
Módulo 7 - Estruturas Dinâmicas - Classes e objetos



- Uma classe é composta por 3 partes:
  - Por que usar propriedades?
    - ✅ **Protegem atributos** para evitar modificações diretas.
    - ✅ Permitem modificar a lógica de acesso a um atributo.
    - ✅ Mantêm a sintaxe simples e elegante.

Python

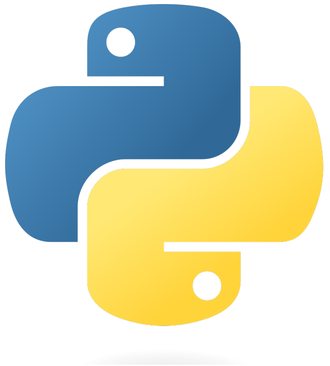
Módulo 7 - Estruturas Dinâmicas - Classes e objetos



- Uma classe é composta por 3 partes:
  - Métodos / Funções: As funções dentro de uma classe são chamadas de métodos.
  - Os métodos definem **comportamentos** que a classe vai manter.
  - No exemplo da Pessoa, comportamentos seriam falar, comer, dormir, etc.

Python

Módulo 7 - Estruturas Dinâmicas - Classes e objetos



- Um **objeto** é uma **instância de uma classe**.
- Ou seja, quando criamos um objeto, estamos a criar **um elemento real** baseado no modelo definido pela classe.
- 📌 Exemplo no Mundo Real:
  - **Classe** → “Carro” (define que todo carro tem cor, marca, modelo).
  - **Objeto** → Um carro específico, como um “Toyota vermelho”.
  - Podemos criar vários **objetos** diferentes (um “Ford azul”, um “BMW preto”), mas todos seguem o mesmo **modelo (classe)**.

Python

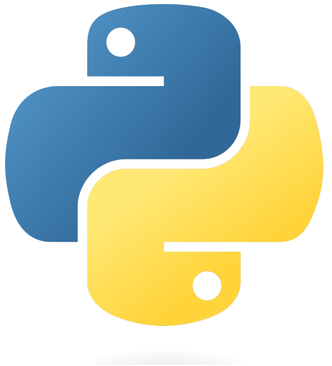
Módulo 7 - Estruturas Dinâmicas - Classes e objetos

- Características de um Objeto
- ✅ **Tem atributos** → Representam as suas características (ex.: cor, marca, modelo).
- ✅ **Tem métodos** → Definem os comportamentos do objeto (ex.: acelerar, travar).
- ✅ **É único** → Cada objeto pode ter valores diferentes para os seus atributos.

```python
# Definição da classe Carro
class Carro:
    def __init__(self, marca, cor):  # Construtor
        self.marca = marca
        self.cor = cor

    def buzinar(self):  # Método do objeto
        print("BEEP BEEP!")

# Criando objetos (instâncias da classe)
carro1 = Carro("Toyota", "Vermelho")
carro2 = Carro("Ford", "Azul")

# Acedendo a atributos e métodos dos objetos
print(carro1.marca)  # Toyota
print(carro2.cor)    # Azul

carro1.buzinar()  # BEEP BEEP!
```

Python

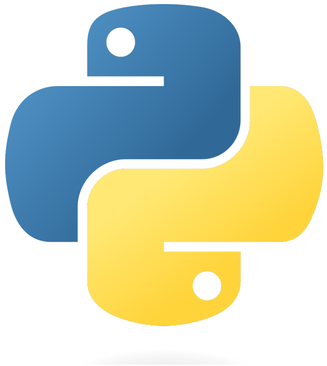
Módulo 7 - Estruturas Dinâmicas - Nós



- O que é um Nó?
- Antes de entrarmos nas estruturas dinâmicas, precisamos entender o conceito fundamental que todas elas compartilham: o **nó**.
- Definição de Nó:
  - Um **nó** é um elemento fundamental em estruturas dinâmicas. Ele contém **dois ou mais componentes principais**:
    - **Valor/Dado** → A informação armazenada no nó (exemplo: um número, texto, um dicionário etc.).
    - **Ponteiro/Referência** → Um link que aponta para o próximo nó (ou para nós relacionados).

Python

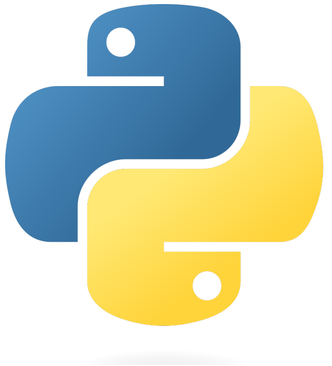
Módulo 7 - Estruturas Dinâmicas - Nós



- Cada nó:
  - Guarda um valor (exemplo: 10).
  - Aponta para o próximo nó (→).
  - O último nó aponta para **None**, indicando o fim da estrutura.

Python

Módulo 7 - Estruturas Dinâmicas - Pilha

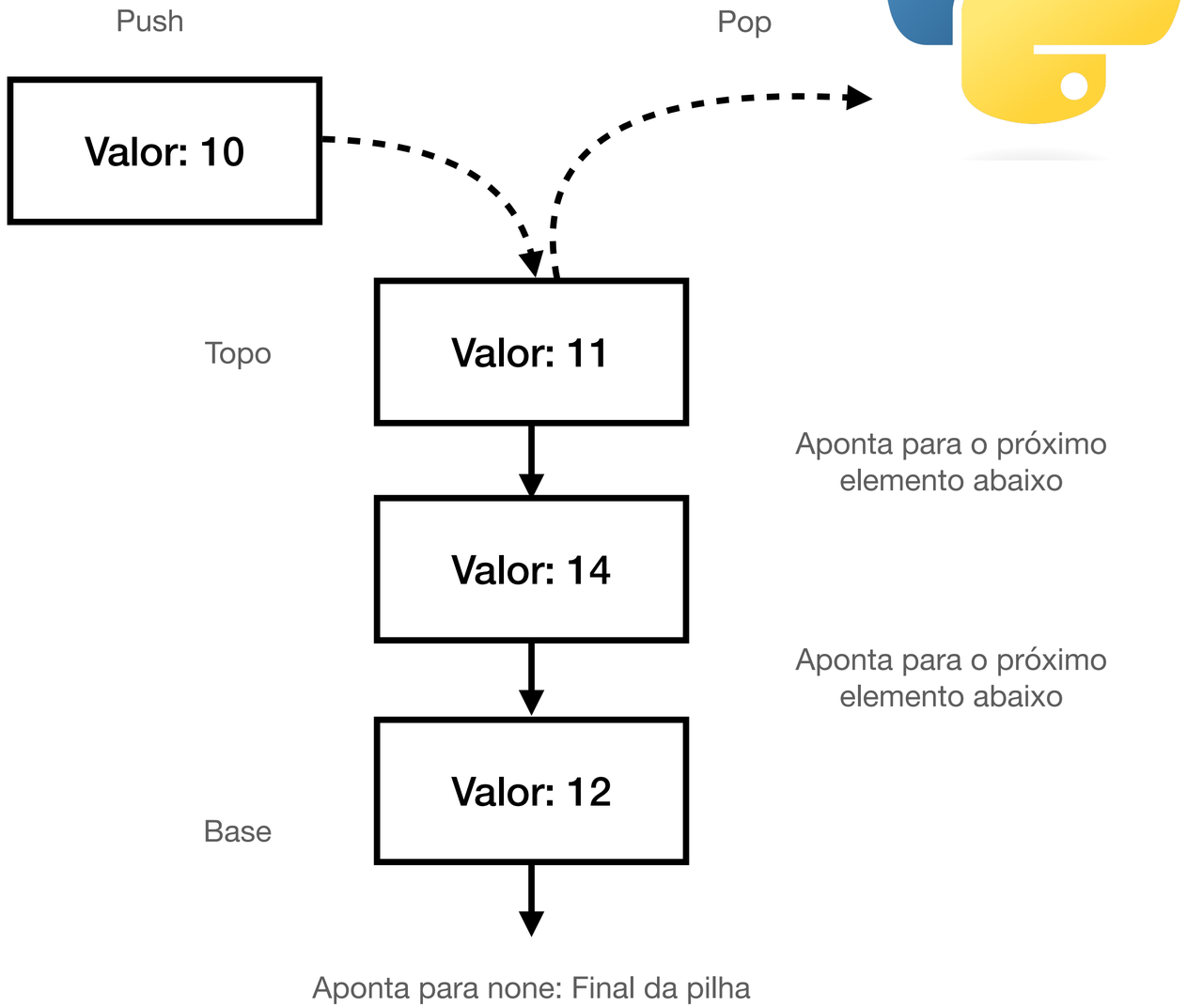


- Uma pilha, também chamada de Stack, é uma estrutura dinâmica onde os elementos são inseridos e removidos seguindo o princípio **LIFO (Last In, First Out)**.
- Ou seja, **o último elemento inserido é o primeiro a ser removido**.
- Exemplo do Mundo Real
  - Uma pilha de pratos: Se colocarmos um prato novo no topo da pilha, ele será o **primeiro a ser retirado**.

Python

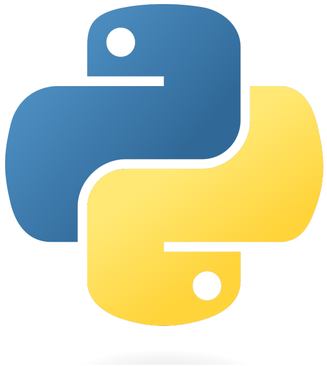
Módulo 7 - Estruturas Dinâmicas - Pilha

- Operações Principais:
  - **Push** → Adicionar um elemento ao topo da pilha.
  - **Pop** → Remover o elemento do topo da pilha.
  - **Top/Peek** → Consultar o elemento no topo sem removê-lo
  - No exemplo ao lado, se fizermos o pop, o nó com o valor 11 vai ser removido. Se fizermos o push com o valor 10, ele será inserido no topo da pilha.



Python

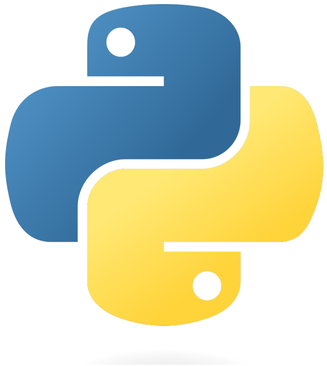
Módulo 7 - Estruturas Dinâmicas - Pilha



```python
class No:
    def __init__(self, valor):
        self.valor = valor
        self.proximo = None  # Aponta para o próximo nó abaixo
class Pilha:
    def __init__(self):
        self.topo = None # Inicialmente a pilha está vazia

    def push(self, valor): # Adiciona um valor no topo da pilha
        novo_no = No(valor) # Cria um novo nó
        novo_no.proximo = self.topo # Coloca o novo nó no topo a apontar para o nó que antes era o topo
        self.topo = novo_no # Atualiza o topo para o novo nó
    def pop(self): # Remove o valor do topo da pilha
        if self.topo is None:
            return None  # Pilha vazia
        valor_removido = self.topo.valor # Guarda o valor do topo
        self.topo = self.topo.proximo # Atualiza o topo para o nó abaixo
        return valor_removido # Retorna o valor removido
        # Neste momento o topo aponta para o nó que antes era o segundo da pilha

# Criar uma pilha vazia
pilha = Pilha()

# Adicionar elementos
pilha.push(1) # adiciona o 1
pilha.push(2) # adiciona o 2 e coloca o nó 2 a apontar para o nó 1
pilha.push(3) # adiciona o 3 e coloca o nó 3 a apontar para o nó 2
# Neste momento a pilha está assim:
# 3 -> 2 -> 1 -> None
# Remover elementos
print(pilha.pop()) # Remove o 3
# Neste momento a pilha está assim:
# 2 -> 1 -> None
```

Python

Módulo 7 - Estruturas Dinâmicas - Fila

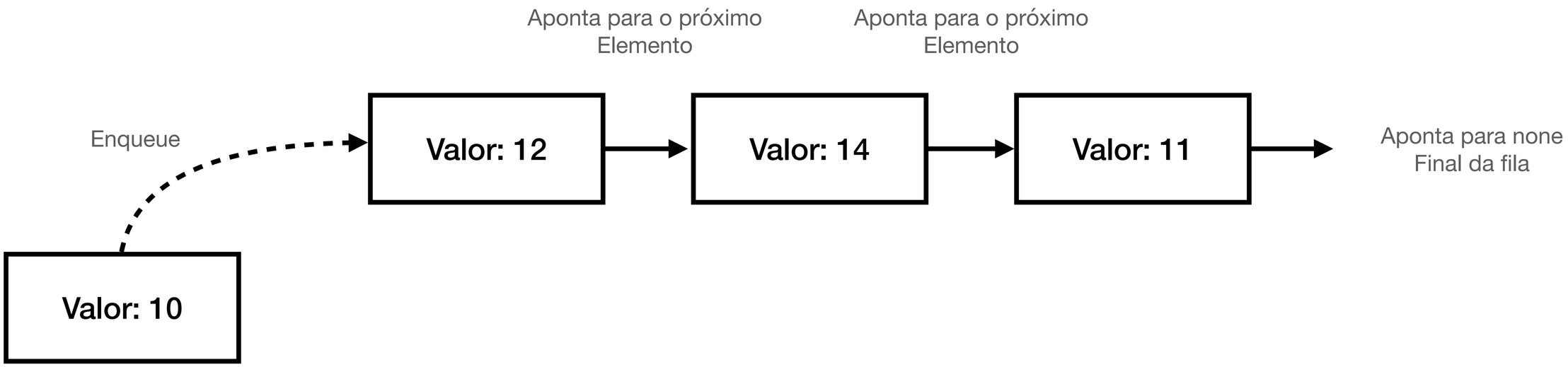


- Uma **fila**, também chamada de queue, é uma estrutura dinâmica onde os elementos seguem o princípio **FIFO (First In, First Out)**.
- Ou seja, **o primeiro elemento a ser inserido é o primeiro a ser removido**.
- Exemplo do Mundo Real
  - Uma fila de supermercado: A primeira pessoa a entrar na fila é a **primeira a ser atendida**.

Python

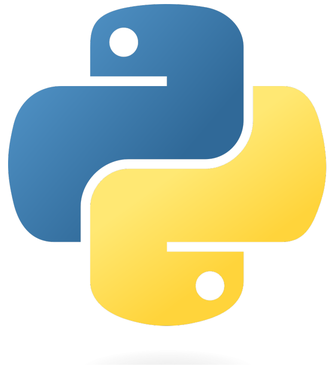
Módulo 7 - Estruturas Dinâmicas - Fila

- Operações Principais
  - **Enqueue** → Adicionar um elemento ao final da fila.
  - **Dequeue** → Remover o primeiro elemento da fila.



Python

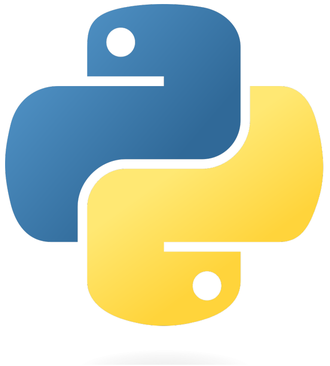
Módulo 7 - Estruturas Dinâmicas - Fila



```python
class No:
    def __init__(self, valor):
        self.valor = valor
        self.proximo = None  # Aponta para o próximo nó

class Fila:
    def __init__(self):
        self.inicio = None  # Primeiro elemento
        self.fim = None  # Último elemento
        # Se a fila está vazia, inicio e fim são None

    def enqueue(self, valor):
        novo_no = No(valor) # Cria um novo nó
        if self.fim: # Se a fila não está vazia
            self.fim.proximo = novo_no # O nó que antes era o último passa a apontar para o novo nó
        self.fim = novo_no # Atualiza o fim para o novo nó
        if self.inicio is None: # Se a fila está vazia
            self.inicio = novo_no # O início passa a ser o novo nó
            # Neste caso, o novo nó é o único nó da fila e como tal é o início e o fim

    def dequeue(self):
        if self.inicio is None: # Se a fila está vazia
            return None  # Fila vazia - não há nada para remover
        valor_removido = self.inicio.valor # Guarda o valor do nó que está no início
        self.inicio = self.inicio.proximo # Atualiza o início para o nó seguinte
        if self.inicio is None: # Se a fila ficou vazia
            self.fim = None # Atualiza o fim para None
        return valor_removido # Retorna o valor removido

# Criar uma fila vazia
fila = Fila()
# Adicionar elementos
fila.enqueue(1) # adiciona o 1 e como a fila está vazia, o início e o fim passam a ser o nó 1
fila.enqueue(2) # adiciona o 2 e coloca o nó 2 a apontar para o nó 1
# o início é o nó 1 e o fim é o nó 2
fila.enqueue(3) # adiciona o 3 e coloca o nó 3 a apontar para o nó 2
# o início é o nó 1 e o fim é o nó 3
```

Python

Módulo 7 - Estruturas Dinâmicas - Lista ligada (Linked List)

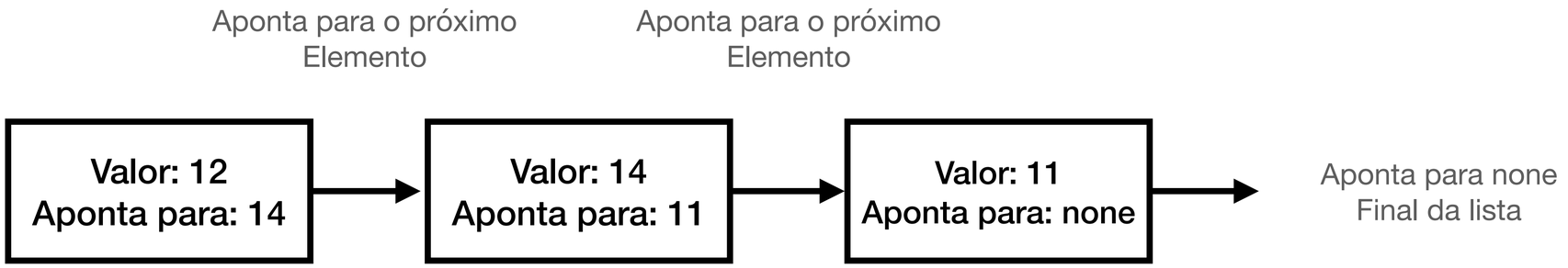


- Uma **lista ligada** é uma coleção de **nós** onde cada nó contém um valor e um ponteiro para o próximo nó.
- Diferente de uma lista normal, a lista ligada **não usa índices fixos**, tornando-a mais eficiente para inserções e remoções.
- Comparando com filas, a lista ligada pode adicionar ou remover um elemento em qualquer parte da estrutura, enquanto que uma fila só pode adicionar no fim e remover no início.

Python

Módulo 7 - Estruturas Dinâmicas - Lista ligada (Linked List)

- Tipos de Listas Ligadas:
  - **Simplesmente Ligada** → Cada nó aponta apenas para o próximo.
  - **Duplamente Ligada** → Cada nó aponta para o próximo e para o anterior.
  - **Circular** → O último nó aponta para o primeiro.



Python

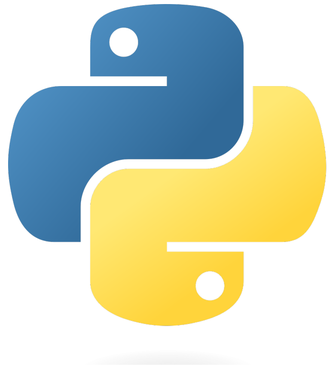
Módulo 7 - Estruturas Dinâmicas - Lista ligada (Linked List)

- Como devem ter reparado, listas ligadas, filas e listas normais são muito parecidas à primeira vista. No entanto têm diferenças significativas

Critério	Lista (Python)	Lista Ligada	Fila
Estrutura Interna	Array dinâmico (vetor)	Nós ligados por ponteiros	Nós ligados por ponteiros (FIFO)
Acesso a Elementos	acesso direto por índice	Percorre um a um. Não é possível aceder diretamente a um elemento pois não tem índice.	Percorre um a um. Não é possível aceder diretamente a um elemento pois não tem índice.
Inserção no Início	Desloca todos os elementos	Ajusta o ponteiro	Insere na cabeça da fila
Inserção no Fim	—	Tem que percorrer até ao último nó	Enqueue no final
Remoção do Início	Desloca todos os elementos	Ajusta o ponteiro	Dequeue
Remoção do Fim	—	Tem que percorrer até ao penúltimo nó	Precisa de reorganizar a fila
Remoção / Inserção a meio	Desloca todos os elementos	Tem que percorrer até ao elemento anterior ao que vai ser removido ou anterior à posição onde vai entrar o novo elemento	Impossível
Uso de Memória	Mais eficiente (guarda elementos sequencialmente)	Mais consumo (cada nó armazena um ponteiro extra)	Mais consumo (como uma lista ligada, mas com operações FIFO)
Flexibilidade	Estrutura fixa (mas pode crescer)	Crescimento dinâmico	Crescimento dinâmico, mas com restrições de ordem
Melhor para	Acesso rápido por índice	Inserções e remoções frequentes	Processamento em ordem FIFO (ex.: filas de espera)

Python

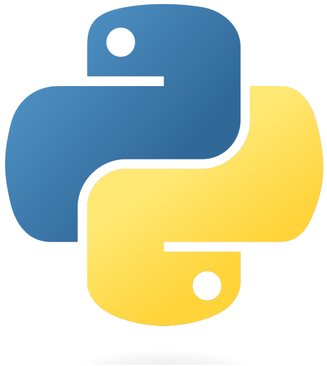
Módulo 7 - Estruturas Dinâmicas - Lista ligada (Linked List)



```python
class No:
    def __init__(self, valor):
        self.valor = valor
        self.proximo = None  # Aponta para o próximo nó

class ListaLigada:
    def __init__(self):
        self.inicio = None # Inicializa a lista vazia

    def inserir_inicio(self, valor):
        novo_no = No(valor) # Cria um novo nó
        novo_no.proximo = self.inicio # Aponta para o início da lista
        self.inicio = novo_no # Atualiza o início da lista com o novo nó

    def imprimir(self):
        atual = self.inicio # Começa do início da lista
        # O atual é um nó. E enquanto o atual.proximo não for None
        # significa que não chegou no final da lista
        # Então, imprime o valor do nó e atualiza o atual para o próximo nó
        # Até que o atual seja None
        while atual:
            print(atual.valor, end=" -> ")
            atual = atual.proximo
        print("None")

# Testando a lista ligada
lista = ListaLigada()

# Inserindo elementos
lista.inserir_inicio(1) # 1 -> None
lista.inserir_inicio(2) # 2 -> 1 -> None
lista.inserir_inicio(3) # 3 -> 2 -> 1 -> None
```

Python

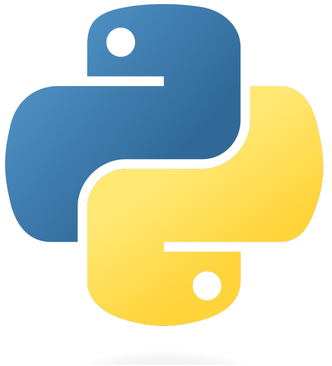
Módulo 7 - Estruturas Dinâmicas - Árvores binárias



- Uma **árvore binária** é uma estrutura hierárquica onde cada nó pode ter, no máximo, **dois filhos**: **esquerdo** e **direito**.
- Cada nó superior tem dois ponteiros (referências) em que um aponta para o filho da esquerda e outro para o filho da direita.
- Propriedades das Árvores Binárias
  - O **nó raiz** é o topo da árvore.
  - Cada nó pode ter zero, um ou dois filhos.
  - **Subárvores** representam os filhos de cada nó.

Python

Módulo 7 - Estruturas Dinâmicas - Árvores binárias

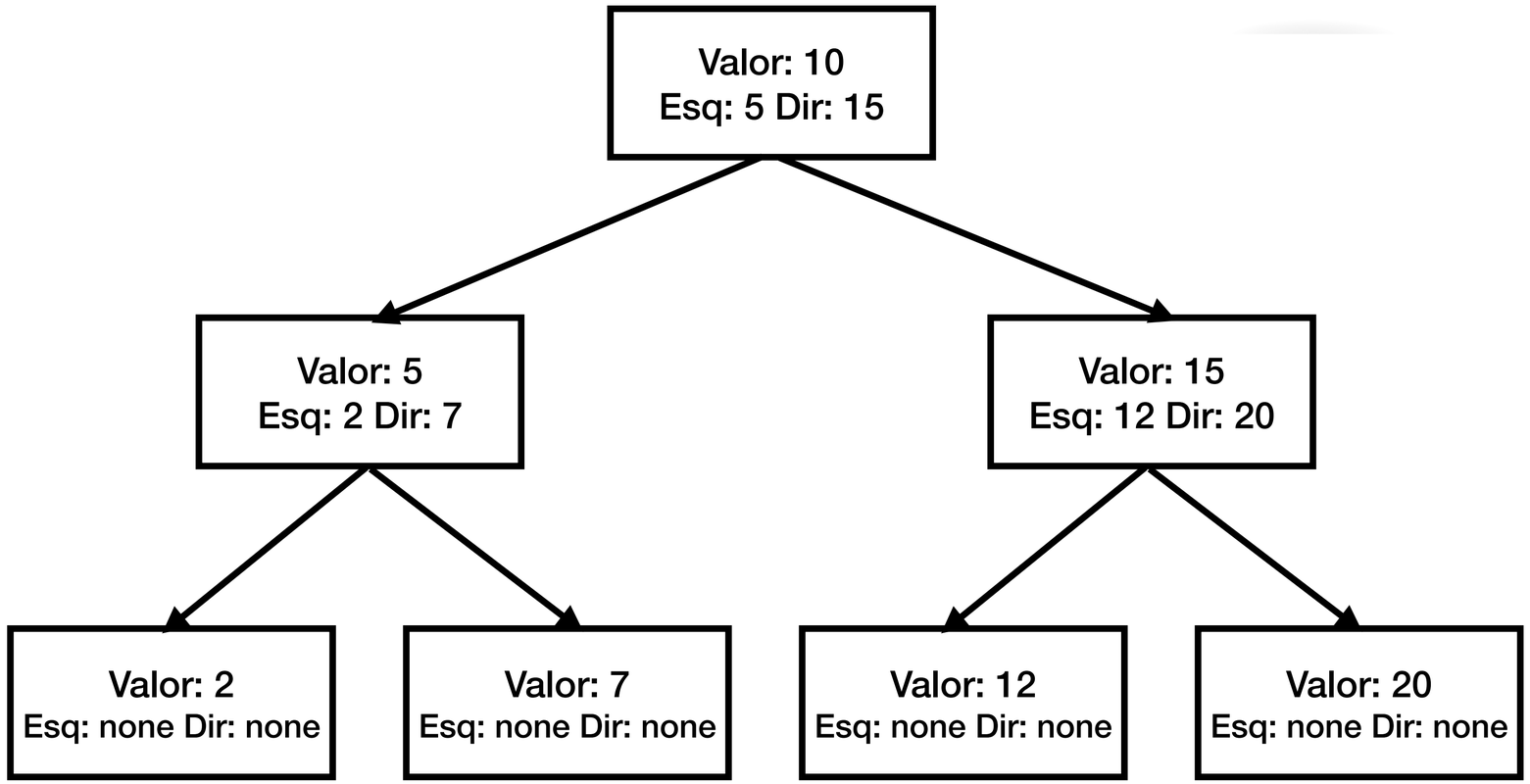


- Vamos supor uma árvore binária para organizar os números 2, 5, 7, 10, 12, 15 e 20.
- Se acharmos o meio vamos verificar que é o número 10. Logo esse será o nó raíz.
- Depois começamos a percorrer os números que queremos introduzir na árvore. Começamos a percorrer os números menores do que 10 para os colocarmos à sua esquerda. Achamos o seu meio. Temos o 2, 5 e 7. O seu meio é o 5.
- Logo o filho da esquerda do 10 será o 5. Os filhos do 5 serão o 2 (esquerda) e o 7 (direita).
- Fazemos o mesmo para os números maiores do 10 e para o lado direito. Entre 12, 15 e 20, o meio é o 15. Logo o filho do lado direito do 10 é o 15.
- De seguida, colocamos o 12 (que é menor do que 15) como o seu filho do lado esquerdo e o 20 como seu filho do lado direito.

Python

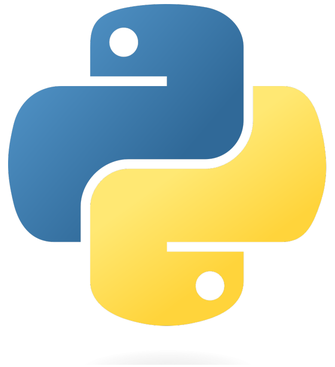
Módulo 7 - Estruturas Dinâmicas - Árvores binárias

- À esquerda do 10, estão os menores valores do que 10. À direita os maiores.
- À esquerda de 5 estão os menores valores do que 5. À direita os maiores
- Etc.
- **Isto significa que o tempo de busca por um nó é reduzido pelo menos para metade. Pois só vamos pesquisar numa das metades da árvore principal.**



Python

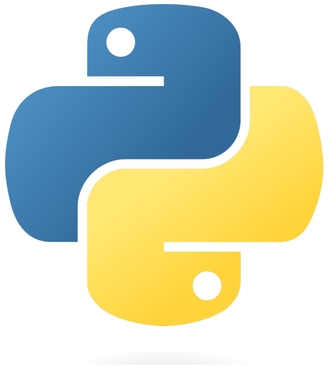
Módulo 7 - Estruturas Dinâmicas - Árvores binárias



```python
class No:
    def __init__(self, valor):
        self.valor = valor
        self.esquerda = None
        self.direita = None

class ArvoreBinaria:
    def __init__(self):
        self.raiz = None # Inicializa a raiz da árvore

    def inserir(self, valor):
        if self.raiz is None: # Se a raiz for nula, insere o valor na raiz
            self.raiz = No(valor) # Cria um novo nó com o valor
        else: # Se a raiz não for nula, chama o método de inserção recursivo
            self._inserir_recursivo(self.raiz, valor)

    # Método de inserção recursivo
    # Para inserir um valor na árvore, é necessário comparar o valor a ser inserido com o valor do nó atual
    # Se o valor a ser inserido for menor que o valor do nó atual, o valor deve ser inserido na subárvore esquerda
    # Se o valor a ser inserido for maior que o valor do nó atual, o valor deve ser inserido na subárvore direita
    def _inserir_recursivo(self, no, valor):
        if valor < no.valor:
            if no.esquerda is None:
                no.esquerda = No(valor)
            else:
                self._inserir_recursivo(no.esquerda, valor)
        else:
            if no.direita is None:
                no.direita = No(valor)
            else:
                self._inserir_recursivo(no.direita, valor)

# Cria uma árvore binária
arvore = ArvoreBinaria()

# Insere valores na árvore
arvore.inserir(3) # Insere o valor 3 na raiz
arvore.inserir(7) # Insere o valor 7 na subárvore direita
arvore.inserir(1) # Insere o valor 1 na subárvore esquerda
```

Python

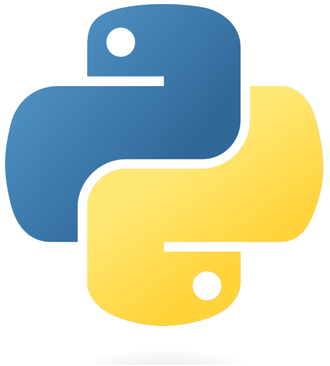
Módulo 7 - Todas as estruturas



- Com este módulo demos 8 estruturas de dados: listas, sets, tuplas, dicionários, pilhas, filas, listas ligadas e árvores.
- Pode ficar confuso saber qual usar e quais as vantagens e desvantagens de cada uma.
- Vamos resumir as vantagens e desvantagens das seis principais estruturas para armazenar grandes quantidades de dados. (todas menos sets e tuplas)

Python

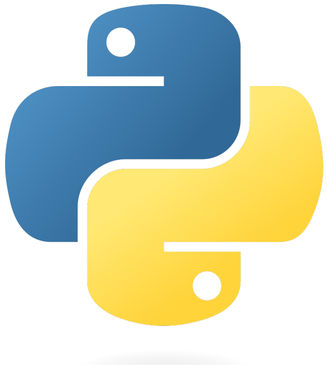
Módulo 7 - Todas as estruturas



- Listas (Arrays Dinâmicos em Python)
  - 📌 Estrutura baseada em arrays dinâmicos, permitindo acesso direto aos elementos.
    - ✅ Vantagens
      - ✔ **Acesso Rápido** → Podemos aceder a qualquer elemento diretamente pelo índice.
      - ✔ **Uso Eficiente de Memória** → Elementos são armazenados de forma contínua.
      - ✔ Métodos embutidos em Python → Suporte a funções como .append(), .pop(), .insert().
    - ❌ Desvantagens
      - ✖ Inserções e Remoções Ineficientes no Meio → Requer deslocamento dos elementos.
      - ✖ **Pré-alocação de Memória** → Quando cresce, pode precisar realocar todos os elementos.
    - 📌 Quando Usar?
      - 🔷 Quando precisas aceder a elementos rapidamente.
      - 🔷 Quando não há muitas inserções ou remoções no meio da estrutura.

Python

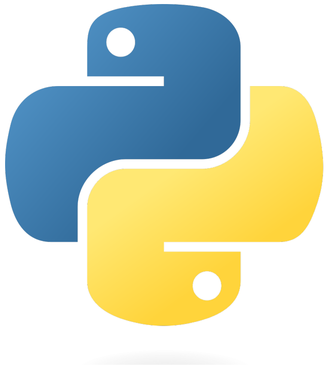
Módulo 7 - Todas as estruturas



- Listas Ligadas
  - 📌 Cada nó tem um valor e um ponteiro para o próximo nó, permitindo crescimento dinâmico.
    - ✅ Vantagens
      - ✔ **Inserção e Remoção Eficientes** → Não há necessidade de deslocar elementos.
      - ✔ **Uso Flexível de Memória** → Cresce conforme necessário, sem realocação.
      - ✔ **Útil para grandes estruturas de dados** → Evita problemas de alocação contínua como em arrays.
    - ❌ Desvantagens
      - ✖ **Acesso Lento** → Para aceder a um elemento, precisamos percorrer a lista.
      - ✖ **Maior Uso de Memória** → Cada nó precisa armazenar um ponteiro extra para o próximo nó.
    - 📌 Quando Usar?
      - 🔷 Quando precisas fazer muitas inserções e remoções, especialmente no início ou meio da estrutura.
      - 🔷 Quando o tamanho da estrutura muda frequentemente.

Python

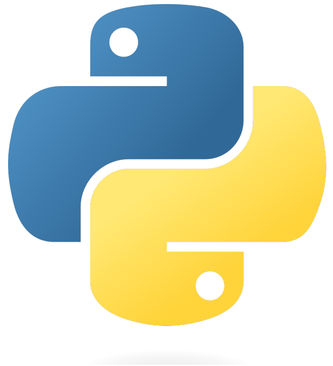
Módulo 7 - Todas as estruturas



- Pilhas (Stacks)
  - 📌 Segue o princípio LIFO (Last In, First Out): o último elemento a entrar é o primeiro a sair.
    - ✅ Vantagens
      - ✔ **Inserção e Remoção Rápidas** → Sempre ocorre no topo da pilha.
      - ✔ **Simples de Implementar** → Apenas um ponteiro para o topo é necessário.
      - ✔ **Útil para Operações Recursivas e Backtracking** → Como chamadas de funções, desfazer ações (Ctrl + Z).
    - ❌ Desvantagens
      - ✖ **Acesso Restrito** → Só conseguimos aceder ao topo da pilha.
      - ✖ **Não é Ideal para procurar Elementos** → Se quiser aceder a um elemento específico, precisaríamos remover todos os que estão acima dele.
    - 📌 Quando Usar?
      - 🔷 Para **desfazer ações** (ex.: histórico de edições).
      - 🔷 Para **resolver problemas recursivos** (ex.: cálculos matemáticos, árvores).

Python

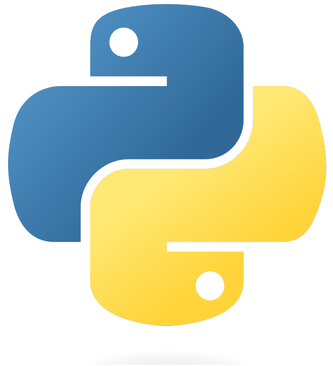
Módulo 7 - Todas as estruturas



- Filas (Queues)
  - 📌 Segue o princípio FIFO (First In, First Out): o primeiro a entrar é o primeiro a sair.
    - ✅ Vantagens
      - ✔ **Inserção e Remoção Rápidas** → Ocorre sempre no início e no fim da fila.
      - ✔ **Óptimo para Processos que Precisam Ser Atendidos por Ordem** → Exemplo: impressão de documentos, filas de atendimento.
      - ✔ **Útil para Algoritmos de Processamento** → Como procura em largura (BFS em grafos).
    - ❌ Desvantagens
      - ✖ **Acesso Restrito** → Só podemos remover do início e adicionar no fim.
      - ✖ **Não é Ideal para procurar Elementos Aleatórios** → Não podemos aceder diretamente um elemento como numa lista.
    - 📌 Quando Usar?
      - 🔷 Para processos onde a ordem de chegada é importante (ex.: filas de banco).
      - 🔷 Para **controlar tarefas em tempo real**

Python

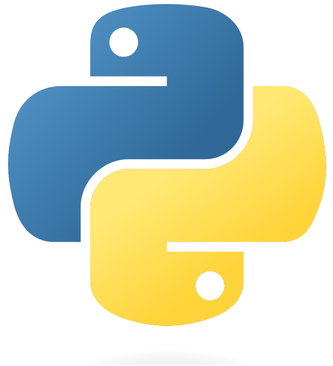
Módulo 7 - Todas as estruturas



- Árvore Binária
  - 📌 Uma estrutura hierárquica onde cada nó pode ter no máximo dois filhos (esquerda e direita).
    - **Cada nó** contém um valor e apontadores para seus filhos.
    - A subárvore esquerda contém valores menores do que o nó pai.
    - A subárvore direita contém valores maiores do que o nó pai.
    - ✅ Vantagens
      - ✔ **Busca eficiente** → Melhor do que listas ou listas ligadas para encontrar elementos.
      - ✔ **Inserção e remoção relativamente rápidas** → Melhor que listas ordenadas, pois não exige deslocamentos.
      - ✔ **Útil para organização de dados** → Como sistemas de ficheiros, bases de dados e estruturas de indexação.
      - ✔ **Boa para ordenação** → Permite percursos **em ordem** para recuperar dados ordenados automaticamente.

Python

Módulo 7 - Todas as estruturas



- Árvore Binária
  - ❌ Desvantagens
    - ✖ **Pode ficar desbalanceada** → Se a árvore não for equilibrada, pode ter desempenho pior.
    - ✖ **Implementação mais complexa** → Precisa de mais código do que listas ou filas.
    - ✖ **Uso de memória maior** → Cada nó armazena um valor e duas referências (esquerda e direita).
    - 📌 Quando Usar?
      - 🔷 Quando precisas armazenar dados ordenados e buscas rápidas (ex.: índices de bases de dados).
      - 🔷 Quando precisas de uma estrutura hierárquica (ex.: organização de ficheiros).
      - 🔷 Quando precisas de uma estrutura eficiente para inserções, remoções e buscas.

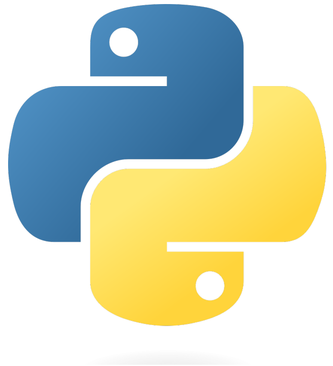
Python

Módulo 7 - Todas as estruturas

- Dicionário (Hash Table em Python)
  - 📌 Estrutura que armazena dados no formato chave-valor, permitindo acesso rápido aos elementos.
    - As chaves são únicas e cada uma aponta para um valor correspondente.
    - Internamente, Python usa tabelas de dispersão (hash tables) para armazenar os dicionários.
    - ✅ Vantagens
      - ✔ **Acesso rápido** → Melhor do que listas para buscar elementos.
      - ✔ **Inserção e remoção rápidas** → Não há deslocamento de elementos como em listas.
      - ✔ Armazena dados de forma estruturada e legível → Permite associar valores a chaves.
      - ✔ **Altamente flexível** → Pode armazenar qualquer tipo de dado como valor (listas, tuplas, outros dicionários).

Python

Módulo 7 - Todas as estruturas



- Dicionário (Hash Table em Python)
  - ❌ Desvantagens
    - ✖ **Consome mais memória** → A tabela hash pré-aloca espaço para otimizar a velocidade.
    - ✖ **Não mantém ordem natural dos elementos**
  - 📌 Quando Usar?
    - 🔷 Quando precisas procurar elementos rapidamente com base numa chave.
    - 🔷 Quando precisas armazenar pares de chave-valor, como configurações ou dados de utilizadores.
    - 🔷 Quando precisas estruturar e agrupar informações relacionadas (ex.: registos de alunos, inventários).

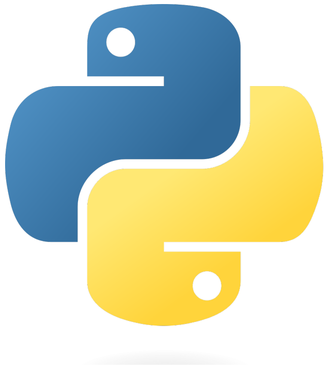
Python

Módulo 7 - Todas as estruturas

Estrutura	Vantagens	Desvantagens	Melhor para
Lista	Acesso rápido (O(1) por índice), fácil de usar	Inserção/remoção no meio lenta (O(n))	Estruturas de acesso rápido
Lista Ligada	Inserção/remoção rápida (O(1)), cresce dinamicamente	Acesso lento (O(n)), mais uso de memória	Estruturas que mudam de tamanho frequentemente
Pilha	Inserção/remoção rápida no topo (O(1)), útil para chamadas recursivas	Acesso restrito (LIFO)	Histórico, backtracking, chamadas de função
Fila	Inserção/remoção eficiente (O(1)), mantém ordem FIFO	Acesso restrito, não permite busca aleatória eficiente	Processamento por ordem, tarefas em tempo real
Árvore Binária	Busca eficiente (O(log n)), ordenação natural	Pode ficar desbalanceada, usa mais memória	Estruturas hierárquicas, busca rápida, organização de dados
Dicionário	Busca muito rápida (O(1) em média), estrutura flexível	Usa mais memória, pode ter colisões	Estruturas de chave-valor, armazenamento de dados estruturados

Python

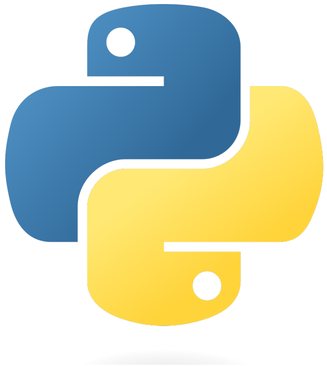
Módulo 7 - Complexidade



- O1? O(log n)?
- Esta é a notação que define a complexidade de uma ação em programação.
- Mas o que é a complexidade?
- A **complexidade de um algoritmo** mede **o tempo e a memória necessários** para a sua execução, dependendo do tamanho da entrada.
- O objetivo é perceber **o quão eficiente** um algoritmo é à medida que os dados aumentam.

Python

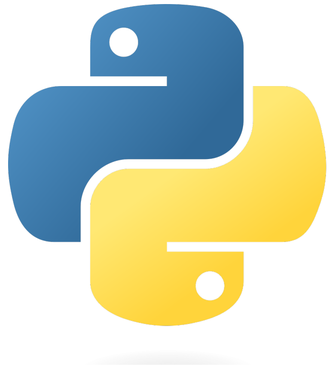
Módulo 7 - Complexidade



- Tipos de Complexidade
  - A complexidade pode ser analisada em dois aspetos principais:
    - Complexidade Temporal
      - Mede **quanto tempo** um algoritmo demora a ser executado.
      - Depende do número de operações realizadas à medida que a entrada cresce.
    - Complexidade Espacial
      - Mede quanto espaço de memória um algoritmo precisa.
      - Inclui variáveis, estruturas de dados, e memória necessária para a execução.

Python

Módulo 7 - Complexidade



- Mas o que é a entrada?
- A entrada refere-se aos dados que vão ser executados num dado algoritmo.
- Vamos supor que temos uma função que recebe uma lista e mostra todos os elementos da lista.
- Aqui a entrada é a lista, pois é nela que estão os dados.

```
def imprimir_lista(lista):
    for elemento in lista:
        print(elemento)
```

Python

Módulo 7 - Complexidade

- Notação Big-O
- A **notação Big-O** é utilizada para descrever **a eficiência de um algoritmo**, indicando **como o tempo de execução cresce** à medida que a entrada aumenta.

Notação	Nome	Exemplo	Explicação
O(1)	Constante	Acesso a lista[i]	O tempo de execução é sempre o mesmo, independentemente do tamanho da entrada.
O(log n)	Logarítmica	Pesquisa Binária	O tempo de execução cresce lentamente, reduzindo a entrada pela metade a cada passo.
O(n)	Linear	Percorrer uma lista	O tempo cresce proporcionalmente ao tamanho da entrada.
O(n log n)	Quasilinear	Algoritmos de ordenação eficientes	Mais rápido que $O(n^2)$, mas mais lento que O(n).
$O(n^2)$	Quadrática	Dois ciclos for aninhados	O tempo cresce rapidamente à medida que a entrada aumenta.
$O(2^n)$	Exponencial	Fibonacci Recursivo	O tempo explode rapidamente, sendo inviável para grandes entradas.

Python

Módulo 7 - Complexidade - Exemplos

- O(1) - Tempo Constante: o tempo de execução **não aumenta** com o tamanho da entrada.

```
lista = [10, 20, 30, 40]
print(lista[2])  # Acesso direto – sempre rápido
```

- O(n) - Tempo Linear: Se a entrada crescer 10 vezes, o tempo de execução **cresce 10 vezes**.

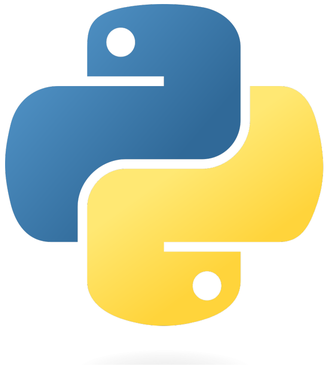
```
lista = [1, 2, 3, 4, 5]
for numero in lista:
    print(numero)
```

- O(n²) - Tempo Quadrático: Cada elemento da lista interage com todos os outros, **multiplicando o número de operações**.

```
for i in range(n):
    for j in range(n):
        print(i, j)  # Executa muitas operações à medida que n cresce
```

Python

Módulo 7 - Complexidade - Estruturas de dados



Estrutura	Acesso	Pesquisa	Inserção	Remoção
Lista (Array Dinâmico)	O(1)	O(n)	O(1) (fim) / O(n) (meio/início)	O(n) (meio/início) / O(1) (fim)
Lista Ligada	O(n)	O(n)	O(1) (início) / O(n) (fim)	O(1) (início) / O(n) (fim)
Pilha (Stack)	O(n)	O(n)	O(1) (push - topo)	O(1) (pop - topo)
Fila (Queue)	O(n)	O(n)	O(1) (enqueue - fim)	O(1) (dequeue - início)
Árvore Binária	O(log n) (balanceada) / O(n) (desequilibrada)	O(log n) / O(n)	O(log n) (média)	O(log n) (média)
Dicionário (Hash Table)	O(1) (média) / O(n) (pior caso - colisões)	O(1) (média) / O(n) (pior caso)	O(1) (média)	O(1) (média)

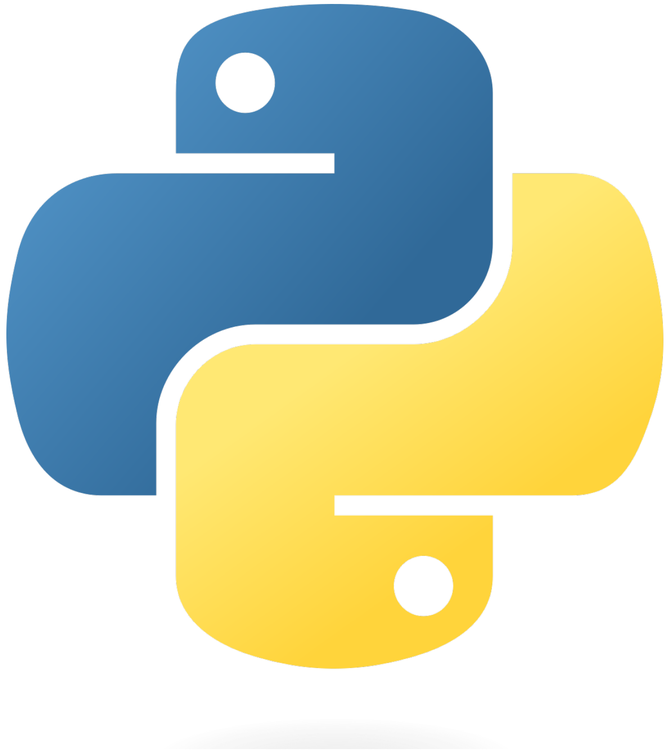
Python

Módulo 7 - Complexidade - Estruturas de dados

- **Listas (Arrays)** são rápidas para **acesso direto (O(1))**, mas lentas para inserção/remoção no meio (**O(n)**).
- Listas Ligadas são mais eficientes para inserção/remoção no início (O(1)), mas têm acesso lento (O(n)).
- Pilhas e Filas são estruturas especializadas com operações rápidas no topo/início (O(1)), mas acesso e pesquisa lentos (O(n)).
- **Árvores Binárias** oferecem **busca eficiente (O(log n))** se estiverem balanceadas, mas podem degradar para **O(n)** se ficarem desbalanceadas.
- Dicionários são as melhores para busca, inserção e remoção (O(1)), mas podem degradar para O(n) em casos raros (colisões).

Linguagens de Programação

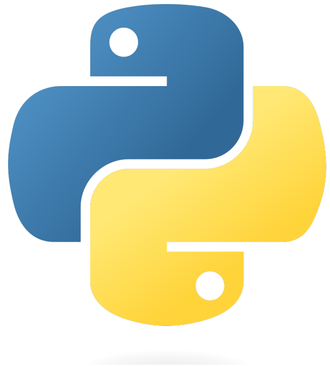
Módulo 7 - Estruturas Dinâmicas



Exercícios

Python

Exercícios



7.

7.1. Calcula a sequência binária dos seguintes números:

7.1.1. - 10

7.1.2. - 25

7.1.3. - 100

7.1.4. - 560

7.1.5. - 788

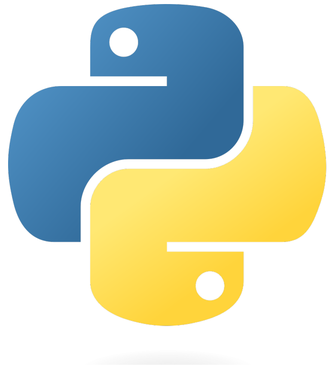
7.1.6. - 1098

7.1.7. - 2089

7.1.8.- 7889

Python

Exercícios



7.

7.2.A que número corresponde as seguintes sequências binárias:

7.2.1. - 1001101

7.2.2. - 10010000101

7.2.3. - 101011

7.2.4. - 1111111111

7.2.5. - 100010100111001

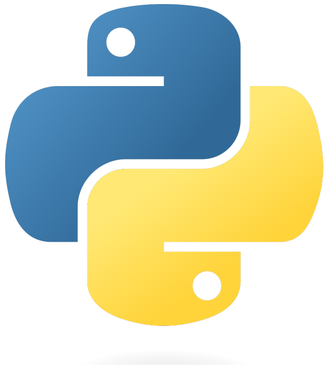
7.2.6. - 10001000100011110001111000111

7.2.7. - 100101111001

7.2.8.- 1000000

Python

Exercícios



7.

7.3.Considerando o código ASCII no slide seguinte, diz que palavras estão nas sequências binárias:

7.3.1. - 01000010 01101111 01101101 00100000 01100100 01101001 01100001

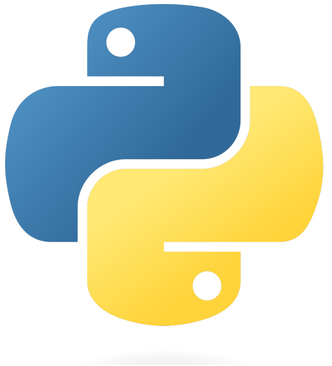
7.3.2. - 01000010 01101111 01100001 00100000 01101110 01101111 01101001 01110100 01100101

7.3.3. - 01000101 01110011 01110100 11100001 00100000 01110101 01101101 00100000 01100010 01100101 01101100 01101111 00100000 01110100 01100101 01101101 01110000 01101111

7.3.4. - 01000011 01101000 01101111 01100011 01101111 01101100 01100001 01110100 01100101

Python

Exercícios



Letra	ASCII	Letra	ASCII	Letra	ASCII	Letra	ASCII	Letra	ASCII	Letra	ASCII
A	65	a	97	À	192	à	224	Á	193	á	225
B	66	b	98	Â	194	â	226	Ã	195	ã	227
C	67	c	99	Ä	196	ä	228	Å	197	å	229
D	68	d	100	Æ	198	æ	230	Ç	199	ç	231
E	69	e	101	È	200	è	232	É	201	é	233
F	70	f	102	Ê	202	ê	234	Ë	203	ë	235
G	71	g	103	Ì	204	ì	236	Í	205	í	237
H	72	h	104	Î	206	î	238	Ï	207	ï	239
I	73	i	105	Ñ	209	ñ	241	Ò	210	ò	242
J	74	j	106	Ó	211	ó	243	Ô	212	ô	244
K	75	k	107	Õ	213	õ	245	Ö	214	ö	246
L	76	l	108	Ù	217	ù	249	Ú	218	ú	250
M	77	m	109	Û	219	û	251	Ü	220	ü	252
N	78	n	110	ÿ	255	ß	223				
O	79	o	111	P	80	p	112	Q	81	q	113
R	82	r	114	S	83	s	115	T	84	t	116
U	85	u	117	V	86	v	118	W	87	w	119
X	88	x	120	Y	89	y	121	Z	90	z	122