

Programação em Python

<https://advancedinstitute.ai>



Tipos e Gerência de Memória em Python

Referências e Fontes das Imagens

- ❑ Python '!= ' Is Not 'is not': Comparing Objects in Python
- ❑ Memory management in Python
- ❑ How does Memory Allocation work in Python (and other languages)?
- ❑ Learning Python: Powerful Object-Oriented Programming (Book)
- ❑ Fluent Python (Book)

Objetos Python

- ❑ Noção mais fundamental na programação Python
- ❑ Em Python, dados assumem a forma de objetos:
 - Objetos integrados que o Python fornece
 - Objetos que criamos usando classes Python ou
 - Ferramentas de linguagem externa, como bibliotecas de extensão C
- ❑ Essencialmente apenas partes da memória, com valores e conjuntos de operações associadas;
- ❑ **Tudo é um objeto em um script Python**
 - Números e suas operações (adição, subtração, etc.)

Hierarquia conceitual de objetos

- Programas Python podem ser pensados como sendo compostos por módulos, instruções, expressões e objetos
 - Os programas são compostos por módulos.
 - Módulos contêm declarações.
 - As declarações contêm expressões.
 - As expressões criam e manipulam objetos.
 - e.g., a atribuição `a = 1 + 1`, temos uma declaração contendo uma expressão que cria um objeto.

Referências a objetos

- ❑ Nomes que se referem à **localização específica na memória** de um objeto.
- ❑ Uma variável ou outra referência **não tem tipo intrínseco**.
 - **O objeto ao qual uma referência** está associada em um determinado momento **tem um tipo**
- ❑ **Pode apontar** para objetos de **diferentes tipos** durante a execução de um programa;
- ❑ A existência de uma variável **depende de uma declaração que vincula a variável a** um objeto armazenado em memória.
- ❑ A instrução `del` desvincula uma referência

Removendo referências

```
1 >>> a = 10
2 >>> print(a)
3 10
4 >>> del(a)
5 >>> print(a)
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8   NameError: name 'a' is not defined
```

Verificação de Tipos

- ☐ Utilizar a operação `type()` para checagem do tipo
- ☐ Os **nomes não têm tipos**;
- ☐ Os tipos **vivem com objetos**, não nomes.

```
1 >>> a = 10
2 >>> type(a)
3 <class 'int'>
4 >>> a = "hello world"
5 >>> type(a)
6 <class 'str'>
```

- ☐ **Alteração da referência.**

Tipos em Python

Verificação de Tipos

- ❑ Os objetos sabem que tipo são;
- ❑ Como os objetos conhecem seus tipos, as variáveis não precisam.
- ❑ Tipos também possuem tipos:

```
1 >>> type(type(a))  
2 <class 'type'>  
3 >>> type(type(type(a)))  
4 <class 'type'>
```

Importante

Tudo o que podemos dizer sobre uma variável em Python é que ela faz referência a um objeto específico em um determinado momento.

Tipos de Objetos

- ❑ **Objetos simples** (números, strings, etc)
- ❑ **Objetos do tipo contêiner** (listas, dicionários, etc.)
- ❑ **Instâncias de Classes personalizadas** definidas pelo usuário (instância da classe `Point`, etc.)

Conceitos

- ❑ Variáveis são **entradas em uma tabela de endereços do sistema**.
- ❑ Objetos **são pedaços de memória alocada**.
- ❑ As referências **são ponteiros seguidos automaticamente** de variáveis para objetos.
- ❑ Cada objeto também tem dois campos de cabeçalho padrão:
 - **Designador de tipo**
 - **Contador de referência**
- ❑ Python **reutiliza internamente certos tipos de objetos** imutáveis, como pequenos inteiros e strings;
 - Cada 0 não é realmente um novo pedaço de memória.

Alocação de memória

□ Alocação estática:

- O programa tem memória **alocada em tempo de compilação**;
- e.g., C/C++ *Arrays* com **tamanhos fixos**;
- Memória **não pode ser reutilizada**
- Área de memória chamada **stack** é utilizada

□ Alocação dinâmica:

- Memória alocada em **tempo de execução**
- Aquilo que é *instanciado*
- Espaço de memória pode ser liberado e **reutilizado**;
- Utiliza área chamada **Heap**;

Regiões de Memória

Application Memory

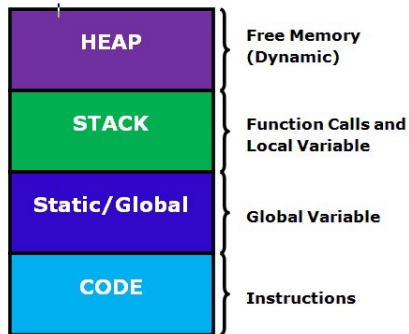


Figure: Regiões de Memória

Garbage Collector

- Processo de liberar memória alocada para um objeto que não está mais em uso;
 - *Memory Leaks*
- “*Reciclagem*” da memória;
- Python usa **contagem de referência** combinada com **detecção de referências cíclicas** para liberar memória não utilizada.
- *Python Memory Manager*
 - Responsável pela alocação e liberação de memória
 - Não necessariamente libera a memória de volta para o seu sistema operacional;

Referências compartilhadas

☐ Quais os valores de L1 e L2 no código abaixo?

```
1 >>> L1 = [1, 2, 3]
2 >>> L2 = L1
3 >>> L1[0] = 0
```

☐ E agora?

```
1 >>> L1 = [1, 2, 3]
2 >>> L2 = L1[:]
3 >>> L1[0] = 0
```

Igualdade vs Identidade

- ❑ Operadores `is` e `==`;
- ❑ O operador `==` compara o **valor ou igualdade** de dois objetos, enquanto o operador `is` verifica se duas variáveis **apontam para o mesmo objeto na memória**.
- ❑ Você pode usar `id()` para verificar a identidade de um objeto:

```
1 >>> my_name = "Raphael"
2 >>> my_surname = "Cobe"
3 >>> id(my_name)
4 139892501805552
5
6 >>> id(my_surname)
7 139892501677360
```


Igualdade vs Identidade

- ❑ Comando `id` retorna o endereço de memória na implementação do CPython;
- ❑ Algumas variáveis são internalizadas no CPython:
 - Valores entre -5 e 256
 - Cada número é **armazenado em um local único e fixo na memória**, o que **economiza memória** para os números inteiros comumente usados.
- ❑ Você pode usar `sys.intern()` para internalizar strings para desempenho;
 - Esta função permite que você **compare seus endereços de memória** em vez de comparar as strings caractere por caractere;
- ❑ O comportamento do operador `==` **pode ser reescrito**;
 - A implementação **padrão é comparar endereços de memória**;



Classes e Objetos em Python

Referências e Fontes das Imagens

- ❑ [Python Object Oriented Programming Tutorial](#)
- ❑ [Python 3 Object Oriented Programming \(Book\)](#)
- ❑ [Learning Python: Powerful Object-Oriented Programming \(Book\)](#)

Classes e Objetos em Python

Definição de Classes

```
1 >>> class MinhaPrimeiraClasse:  
2     ...     pass
```

- ❑ Em Python, a instrução `pass` é uma instrução nula;
- ❑ É usado como um *placeholder* para implementação futura de funções, loops, etc.

Instanciando um objeto a partir de uma classe

```
1 >>> var1 = MinhaPrimeiraClasse()  
2 >>> type(var1)  
3 <class '__main__.MinhaPrimeiraClasse'>  
4 >>> var1.__class__  
5 <class '__main__.MinhaPrimeiraClasse'>
```

Definição de Classes

- ❑ Definem um *namespace* para agrupamento de **atributos**;
- ❑ Ideia parecida com módulos
 - No entanto, classes são declaradas com sentenças e não correspondem a um arquivo em separado;
 - Somente temos uma instância de um dado módulo carregado em memória;
- ❑ São uma espécie de **fábrica para gerar instâncias**;
- ❑ Delimitam estado e comportamento;
 - Atributos e funções que manipulam esses atributos;

Adicionando Atributos

□ Atributos de Classe vs de Instância

```
1 >>> var1.msg1 = "Hello World"
2 >>> var1.msg1
3 'Hello World'
4 >>> MinhaPrimeiraClasse.msg2 = "Ola Mundo"
5 >>> var1.msg2
6 'Ola Mundo'
```

Adicionando Comportamento

```
1 >>> class Point:
2 ...     def reset(self):
3 ...         self.x, self.y = 0, 0
4 >>> p = Point()
5 >>> p.reset()
6 >>> p.x, p.y
7 (0, 0)
```

- ❑ A única diferença entre **métodos** e **funções normais** é que **todos os métodos têm um argumento obrigatório** (por convenção chamado **self**)
- ❑ Uma referência ao objeto em que o método está sendo invocado;
 - **Passado automaticamente** para o método pelo interpretador Python;

Inicializando Objetos

- ❑ Método especial `__init__()`;
 - Chamado quando o **construtor** de uma classe é invocado, e.g. o Método `Point()`
 - Utilizado para inicializar o estado do objeto que se está criando

```
1 >>> class Point:
2 ...     def __init__(self, x=0, y=0):
3 ...         self.x, self.y = x, y
4 ...
5 >>> p = Point(10,20)
6 >>> p.x, p.y
7 (10, 20)
```


Dúvidas?