Aula-01

November 1, 2023

1 Módulo de Programação Python: Introdução à Linguagem

2 Primeiro Encontro

2.1 Apresentação de Python: Sintaxis, tipos de dados e operadores

Objetivo: Introduzir o **Python** como linguagem de programação, configurar o ambiente de desenvolvimento e dar os primeiros passos. Ensinar a sintaxe básica de **Python**, variáveis, tipos de dados básicos e seus operadores. Explorar o uso de tipos de dados avançados em **Python**: listas e tuplas, dicionários e conjuntos.

2.1.1 Uma Introdução Rápida

Concebida no fim dos anos 80 como uma linguagem de script, **Python** se converteu nos últimos anos numa ferramenta essencial para muitos programadores, engenheiros e cientistas. Veja um pouco sobre a história do desenvolvimento de *Python* em A História do Python.

> Fonte: History of the Python language

Python pode ser definida como uma linguagem de programação de alto nível que permite lidar com várias tarefas de programação, como computação numérica, desenvolvimento web, programação de banco de dados, programação de rede, processamento paralelo, etc. Pode-se acrescentar a isto o fato de estar disponível para os sistemas operacionais mais populares, como Windows, Mac e Linux.

O fato de se tratar de uma linguagem interpretada permite que programadores possam testar o código em ambientes interativos, antes de incorporar no programa final, eliminando a necessidade de processos demorados de compilação. Entretanto, muitos programadores procuram Python com base na sua simplicidade e elegância, bem como devido à vantagem de contar com um ecossistema de ferramenta construídas em cima desta poderosa linguagem.

A modo de exemplo pode-se falar do desenvolvimento de aplicações em computação científica e ciência dos dados que são construídas em torno de um grupo de pacotes maduros e úteis:

- **NumPy** : fornece armazenamento e ferramentas de cômputo eficientes para arrays e matrizes de dados;
- **SciPy** : contém uma ampla gama de ferramentas numéricas, como rotinas para integração numérica ou interpolação;
- **Pandas** : fornece um DataFrame, juntamente com um poderoso conjunto de métodos para manipular, filtrar, agrupar e transformar dados.
- **Matplotlib** : disponibiliza uma interface útil para a criação de gráficos e figuras de qualidade para publicação;

- **Scikit-Learn** : fornece um conjunto de ferramentas para a aplicação de algoritmos de aprendizagem de máquina.
- **IPython / Jupyter** : fornece um terminal aprimorado e um ambiente de notebook interativo, muito útil para análises exploratórias, bem como para criação de documentos interativos e executáveis.

Como seu foco está na capacidade de programar mais rapidamente, a velocidade de execução é prejudicada em alguns casos. Um programa Python pode ser até 10 vezes mais lento do que um programa C equivalente, mas conterá menos linhas de código e pode ser programado para lidar facilmente com vários tipos de dados.

> Fonte: Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. P., & Saraiva, J. (2017). Energy efficiency across programming languages: How do energy, time, and memory relate. SLE 2017 - Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, Co-Located with SPLASH 2017, 256–267. DOI

Essa desvantagem no código Python pode ser superada convertendo as partes computacionalmente intensivas do código para C/C++ ou pelo uso apropriado de estrutura de dados e módulos disponíveis.

Para aproveitar o poder deste ecossistema, no desenvolvimento de aplicações em computação científica, primeiro é necessário ter uma determinada familiaridade com a própria linguagem Python. Muitos dos alunos deste curso tem familiaridade com outras linguagens - MATLAB, Pascal, Java, C ++, etc. - pelo que se tentaremos disponibilizar uma revisão breve, mas abrangente desta linguagem.

2.1.2 Como executar código escrito em Python

Um ponto de partida para criar seu ambiente de desenvolvimento pode ser instalar a ferramenta Anaconda

Basicamente dispomos de quatro mecanismos para executar código escrito em **Python**

- O interpretador **Python**: Disponível nos principais **SO** e distribuições **GNU/Linux**, o interpretador pode ser utilizado de forma simples e é bastante conveniente para experimentar pequenos fragmentos de códigos e sequências curtas de operações;
- O interpretador **IPython**: Desenvolvido como um ambiente interativo (**IPython** = *Interactive Python*) acrescenta uma série de recursos importantes ao interpretador;
- Utilizando *scripts*: Para programas maiores e mais complicados é conveniente salvar o código na forma de um arquivo **.py** que, posteriormente, pode ser executado com o interpretador;

  Exemplo

[46]:
```
%run script_001
```

```
c = a + b =  1  +  2  =  3
```

- Utilizando o **Jupyter notebook**: Criado como um híbrido de um terminal interativo com um arquivo de *script*, o *notebook* é uma mistura de código executável, texto, gráficos, e outras ferramentas interativas em um único documento.

Uma revisão rápida da sintaxes de Python Veja o exemplo a seguir:

[47]:
```
import random

# definindo o ponto médio
pontoMedio = 128

# criando duas listas vazias
menores = []; maiores = []

# Separando os números entre menores e maiores
for i in range(10):
    pix = random.randint(0,255)
    if(pix < pontoMedio):
        menores.append(pix)
    else:
        maiores.append(pix)

print("menores:", menores)
print("maiores:", maiores)
```

```
menores: [19, 49, 81, 100]
maiores: [202, 234, 220, 141, 235, 179]
```

Deste pequeno exemplo podemos extrair alguns aspectos iniciais, importantes, da linguagem

Comentários em Python Os comentários são uma parte importante de qualquer linguagem de programação. Comentários em **Python** são indicados pelo jogo da velha (hash - **#**) e qualquer coisa após este símbolo é ignorado pelo interpretador. Desta forma comentários podem estar na forma de uma linha separada ou no final de uma linha de código, como no exemplo a seguir.

[48]:
```
pix = 128       # Operador de atribuição
pix = pix + 2   # Operador de adição
# Operador composto de soma a adição
pix += 3        # equivalente a x = x + 3
```

Quando se deseja incluir um comentário que se extende por várias linhas, podem ser utilizadas *strings* delimitadas com aspas triplas (aspas simples triplas ou aspas duplas triplas) no início e no final do bloco.

[49]:
```
'''
Isto é um comentario
de múltiplas linnas
'''
pix = 5
```

[50]:
```
"""
Isto também é um comentario
de múltiplas linnas
```

```
"""
pix = 128
```

Fim de linha encerra uma declaração Repare que no exemplo anterior é atribuída à variável `pix` o valor `128`. A declaração termina simplesmente com o fim de linha, ao contrário de outras linguagens como **C**, onde um finalizador (`;`) é necessário.

Naqueles casos particulares em que se faz necessário continuar a declaração na próxima linha, pode-se utilizar a barra invertida (`\`) , como no exemplo a seguir.

[51]:
```
pontoMedio = 32 + 64 + 96 \
    + 160 + 192 + 224
pontoMedio = pontoMedio/6
print(pontoMedio)
```

```
128.0
```

A declaração anterior também pode ser implementada em múltiplas linha com a utilização de parênteses. A utilização de parênteses, delimitando uma expressão, permite continuar a declaração na próxima linha.

[52]:
```
pontoMedio = (32 + 64 + 96 +
              160 + 192 + 224)/6
print(pontoMedio)
```

```
128.0
```

Uso de indentação Se prestar atenção no bloco principal do primeiro exemplo pode-se constatar que fazem parte do mesmo uma estrutura de repetição com um for e uma estrutura condicional com um if/else. Trata-se de um conjunto de instruções que podem ser tratadas como uma unidade ou bloco de código. Veja o seguinte exemplo:

[53]:
```
pontoMedio = 0
pix = 0
for i in range(7):
    pix += 32
    if (pix != 128):
        pontoMedio += pix
pontoMedio /= 6
print(pontoMedio)
```

```
128.0
```

Em linguagens como **C**, os blocos de código são definidos por delimitadores específicos como as chaves `{ }`.

Em **Python** os blocos de códigos são demarcados utilizando indentação. Os blocos de código indentados são sempre precedidos por dois pontos (`:`) no final da linha anterior.

Desta forma, em **Python**, a indentação do código não é opcional. Isso torna o código legível.

No entanto, um código com vários loops, ou outras construções que envolvam o uso de blocos sintácticos, aninhados será varias vezes recuado para a direita, dificultando a leitura do código.

Uso de espaço O espaço em branco antes do começo da linha é utilizado para indentação e tem um significado específico na linguagem: linhas indentadas pertencem a um bloco sintáctico.

Já o espaço no interior da linha não tem um significado específico mas pode ser utilizado para melhorar a legibilidade do código.

De forma geral os programadores **Python** recomendam utilizar um espaço antes e depois de cada operador binário e nenhum espaço em operadores unários.

Uso de parênteses Os parênteses podem ser utilizados em diversos contextos. Como em outras linguagens podem ser utilizadas, por exemplo, para agrupar declarações ou operações matemáticas.

Neste contexto a expressão entre parênteses é avaliada antes, sobrepondo-se à ordem definida pela precedência de operadores. Veja a diferença entre as duas expressões utilizadas a seguir:

[54]:
```
# Ordem definida pela precedência dos operadores
pontoMedio = 96 + 160 / 2          # primeiro / e depois +
print("Resultado 1: ", pontoMedio)
# Ordem definida pelos parênteses
pontoMedio = (96 + 160) / 2        # primeiro + e depois /
print("Resultado 2: ", pontoMedio)
```

```
Resultado 1:  176.0
Resultado 2:  128.0
```

Os parênteses são utilizados também para especificar que se deseja chamar uma função. Os parênteses que seguem ao nome da função, contém os argumentos da mesma.

Mesmo nos casos específicos em que a função não precisa de passagem de parâmetros, também são utilizados os parênteses. Veja os exemplos a seguir:

[55]:
```
L = [32, 224, 96, 64, 192, 160]
print("Lista original: ", L)  # chamando à função print
L.sort()                      # chamando à função sort
print("Lista ordenada: ", L)
```

```
Lista original:  [32, 224, 96, 64, 192, 160]
Lista ordenada:  [32, 64, 96, 160, 192, 224]
```

Repare que **print** é uma função que espera argumentos, o que desejamos imprimir. Já **sort** é uma função específica de listas, um método, que não recebe parâmetros.

> **Leituras recomendadas**: Uma leitura interessante para quem quiser utilizar uma sintaxes padronizada é PEP 8 – Style Guide for Python Code

Python: Variáveis e Objetos Trabalhar com variáveis em **Python** é mais simples que outras linguagens de programação.

Para atribuir um valor a uma variável basta apenas colocar o nome da mesma a esquerda seguido pelo operador de atribuição (=) e o valor a ela atribuída a direita.

Em linguagens mais tradicionais, como **C/C++**, as variáveis podem ser pensadas como um receptáculo de tamanho específico, de acordo com o tipo da variável. Desta forma, antes de atribuir qualquer valor, sempre se faz necessário declarar a variável, ou seja, definir qual tipo precisa ser utilizado (o tamanho do receptáculo).

Em **Python** as variáveis podem ser interpretadas como um ponteiro, ou uma referência, para um receptáculo apropriado para armazenar o valor que a ela está sendo atribuída. Desta forma não é necessário "declarar" a variável, o mesmo exigir que uma variável sempre aponte para um mesmo tipo de receptáculo. Veja o exemplo a seguir:

[56]:
```
pix = 128             # pix aponta para um inteiro
print("pix aponta para um inteiro: pix = ", pix)
pix = 'Pixel'          # pix aponta agora para uma string
print("pix aponta agora para uma string: pix = ", pix)
pix = [64, 128, 196]     # pix aponta agora para uma lista
print("pix aponta agora para uma lista: pix = ", pix)
```

```
pix aponta para um inteiro: pix =  128
pix aponta agora para uma string: pix =  Pixel
pix aponta agora para uma lista: pix =  [64, 128, 196]
```

Podemos então definir **Python** como uma linguagem dinamicamente tipada. Esta escolha pode ser elencada como uma das características que fazem da linguagem uma das mais simples e eficientes, em termos de desenvolvimento de código, de se utilizar.

Mas tratar variáveis como ponteiros tem suas especificidades e, quando não tratado de forma adequada, pode ter consequências indesejadas. Veja o exemplo a seguir e tire suas próprias conclusões.

[57]:
```
L1 = [1, 2, 3]          # L1 aponta para uma lista
L2 = L1                 # L2 aponta para a mesma lista que L1
print("Como podemos ver:")
print("L1 = ", L1)
print("e ")
print("L2 = ", L2)
print("apontam para a mesma lista.")
print("Ou seja, se modificarmos a lista referenciamos por L1 ...")
L1.append(4)            # acrescentando um elemanto na lista
print("estamos modificando a lista referenciada por L2 ")
print("L2 = ", L2)
print("Agora, se modoficamos receptáculo para o qual L1 esta apontando ...")
L1 = "Agora é uma string"
print("A variável L2 continua apontando para ")
print("L2 = ", L2)
print("Ao contrario de L1 que")
print("L1 = ", L1)
```

```
Como podemos ver:
L1 =  [1, 2, 3]
```

```
e
L2 =  [1, 2, 3]
apontam para a mesma lista.
Ou seja, se modificarmos a lista referenciamos por L1 …
estamos modificando a lista referenciada por L2
L2 =  [1, 2, 3, 4]
Agora, se modoficamos receptáculo para o qual L1 esta apontando …
A variável L2 continua apontando para
L2 =  [1, 2, 3, 4]
Ao contrario de L1 que
L1 =  Agora é uma string
```

O tratamento de variáveis em **Python** também utiliza o conceito de variáveis **mutáveis** e **imutáveis**.

Para simplificar o tratamento das operações aritméticas, em **Python** os números, strings e outros tipos de variáveis simples são tratados como **imutáveis**. Isto significa que não é possível mudar os valores armazenados na variável mas, apenas, mudar para qual espaço da memória está sendo referenciado. Veja o exemplo a seguir.

[58]:
```
# Exemplo com variáveis imutáveis
pix_1 = 64          # pix_1 referência um inteiro
pix_2 = pix_1       # pix_2 referência o mesmo espaço na memória
print("Inicialmente: ")
print("pix_1 = ", pix_1)
print("e ")
print("pix_2 = ", pix_2)
print("Após incrmentar o valor de pix_1 em 1 temos que ...")
pix_1 += 1          # incrementando pix_1 em 1
print("Agora")
print("pix_1 = ", pix_1)
print("enquanto ")
print("pix_2 = ", pix_2)
```

```
Inicialmente:
pix_1 =  64
e
pix_2 =  64
Após incrmentar o valor de pix_1 em 1 temos que …
Agora
pix_1 =  65
enquanto
pix_2 =  64
```

Todas as variáveis são Objetos Podemos anotar num cantinho acessível do caderno: Em **Python**, uma linguagem de programação orientada a objetos, todas as variáveis são na realidade "referências" a objetos.

Se alguém teve até aqui a ideia de que **Python** é uma linguagem livre de tipos ou fracamente tipada, pode mudar seus conceitos a respeito. Veja o seguinte exemplo que utiliza tipos simples:

[59]:
```
x = 4.5    # x referencia um valor de ponto flutuante
i = 4      # i referencia um valor inteiro
print("O valor de x = ", x)
print("O tipo de x é", type(x))
print("O valor de i = ", i)
print("O tipo de i é", type(i))
# Como instância da classe float x pode ser tratado como ...
print("Qem lembra de números complexos ou imaginarios?")
print(x.real, "+", x.imag, "i")
print("Ainda que o tipo de x continua sendo ", type(x))
print("Podemos testar também os métodos desta clase")
print("Por exemplo: x é um valor inteiro? ", x.is_integer())
x = 4.0
print("Mas se mudamos o valor de x para x = ", x)
print("Agora: x é um inteiro? ", x.is_integer())
```

```
O valor de x =  4.5
O tipo de x é <class 'float'>
O valor de i =  4
O tipo de i é <class 'int'>
Qem lembra de números complexos ou imaginarios?
4.5 + 0.0 i
Ainda que o tipo de x continua sendo  <class 'float'>
Podemos testar também os métodos desta clase
Por exemplo: x é um valor inteiro?  False
Mas se mudamos o valor de x para x =  4.0
Agora: x é um inteiro?  True
```

A afirmação todo em **Python** é objeto pode ser estendida aos atributos e métodos de uma classe que são, por sua vez, objetos com seus próprios atributos métodos. Por exemplo:

[60]:
```
print(type(x.is_integer))    # Uma função (o método da classe)
print(type(x.is_integer()))  # O tipo de retorno da função
print(type(x.real))          # Um atributo da classe
```

```
<class 'builtin_function_or_method'>
<class 'bool'>
<class 'float'>
```

Tipos nativos da dados Vamos começar comentando os tipos nativos de dados mais simples.

Type	Example	Description
`int`	`x = 1`	Números inteiros
`float`	`x = 1.0`	Números de ponto flutuante
`complex`	`x = 1 + 2j`	Números complexos
`bool`	`x = True`	Valores booleanos (True ou False)
`str`	`x = 'abc'`	Cadeias de caracteres, Strings
`NoneType`	`x = None`	Objeto especial indicando null

Falaremos um pouco mais sobre cada um destes tipos

Números Inteiros O tipo numérico mais simples é aquele que representa valores sem caças decimais. Estes valores são representados pela classe `int` que implementa um tipo imutável de dado.

[61]:
```
a = 1
print("A variavel a armazena o valor ", a, " e é de tipo: ", type(a))
```

```
A variavel a armazena o valor  1  e é de tipo:  <class 'int'>
```

O tipo inteiros em **Python** tem características importantes que o destacam de tipos equivalentes em outras linguagens. Para começar: as linguagens imperativas tradicionais tratam os tipos inteiros como valores de tamanho *fixo* e geram condições de *overflow* para valores acima de determinado limite. Em **Python** os inteiros têm tamanho variável e podendo, sem nenhum problema, realizar operações com valores muito grandes.

[62]:
```
2**200
```

[62]:
```
1606938044258990275541962092341162602522202993782792835301376
```

Números de Ponto Flutuante Para tratar de valores com caças decimais são utilizados os números de ponto flutuante. Os números de ponto flutuante podem ser representados em notação decimal tradicional ou em notação científica. Veja os exemplos a seguir.

[63]:
```
x = 0.00001 # Representação tradicional
y = 1e-5    # Mas pode utilizar e ou E para notação científica
print(x == y)

x = 1400000.00
y = 1.4E6  # Pode utilizar e ou E
print(x == y)
```

```
True
True
```

A representação computacional de valores de ponto flutuante impõe limitações na utilização na implementação da aritmética de ponto flutuante. Estas limitações independem da linguagem utilizada e tem a ver com a quantidade de bits utilizados na representação dos valores. Veja o exemplo a seguir e comente:

[64]:
```
0.1 + 0.2 == 0.3
```

[64]:
```
False
```

Veja como estes valores são realmente representados. Para isto vasta agregar algumas caças decimais na hora de imprimir.

[65]:
```
# Veja como são representados os valores anteriores
print("0.1 = {0:.17f}".format(0.1))  # formatação que lembra o printf de c
```

```
print("0.2 = {0:.17f}".format(0.2))
print("0.1 + 0.2 = {0:.17f}".format(0.1 + 0.2))
print("0.3 = {0:.17f}".format(0.3))
```

```
0.1 = 0.10000000000000001
0.2 = 0.20000000000000001
0.1 + 0.2 = 0.30000000000000004
0.3 = 0.29999999999999999
```

Em **Python** os números de ponto flutuante são truncados internamente, por padrão, em 52 bits após o primeiro bit diferente de zero. Devemos então levar em consideração que a aritmética de ponto flutuante é sempre aproximada e testes de igualdade estrita não são recomendados quando envolvidas este tipo de operações.

Veja mais detalhes sobre as operações com valores de ponto flutuante em # Floating Point Arithmetic: Issues and Limitations

Números Complexos Nos casos em que se faz necessário trabalhar com valores do domínio dos números complexos, **Python** apresenta uma classe específica para representar este tipo de valores. Trata-se de números com parte real e imaginárias que podem ser definidos com valores inteiros ou de ponto flutuante.

[66]:
```
lam_1 = complex(1, 2)   # Representação via complex
lam_2 = 1 + 2j;         # Notação para numeros complexos j ou J
print("lam_1 = ", lam_1, " : ", type(lam_1))
print("lam_2 = ", lam_2, " : ", type(lam_2))
print("lam_1 == lam_2 -> ", lam_1 == lam_2)
```

```
lam_1 =  (1+2j)  :  <class 'complex'>
lam_2 =  (1+2j)  :  <class 'complex'>
lam_1 == lam_2 ->  True
```

A classe `complex` implementa uma serie de atributos e métodos com características e operações básicas para tratar números complexos. Veja alguns exemplos das mesmas, assim como do uso de operadores aritméticos

[67]:
```
print("lam_1 = ", lam_1)
print("Parte real de lam_1: ", lam_1.real)
print("Parte imaginária de lam_1: ", lam_1.imag)
print("O complexo conjugado de lam_1: ", lam_1.conjugate())
print("O módulo e valor absoluto de lam_1: ", abs(lam_1))
```

```
lam_1 =  (1+2j)
Parte real de lam_1:  1.0
Parte imaginária de lam_1:  2.0
O complexo conjugado de lam_1:  (1-2j)
O módulo e valor absoluto de lam_1:  2.23606797749979
```

Cadeias de caracteres (Strings) Cadeias de caracteres ou *strins* são amplamente utilizados em diferentes aplicações. Podemos definir uma *string* em **Python** como um conjunto de carac-

teres delimitados utilizando aspas simples ou duplas. Os objetos de tipo *string* representam tipos imutáveis e possuem uma serie de métodos implementados muito uteis. Veja os exemplos a seguir.

[68]:
```
disciplina = "CET1202 - Algoritmos e Programação"
professor = 'Esbel T. Valero Orellana'

# Tamanho da string
print("Tamanho de disciplina: ", len(disciplina))
# transformando maiusculas e minusculas
print("Professor: ", professor.upper())
print("Disciplina: ", disciplina.lower())
print("alguem tem dúvidas?".capitalize())

#Os métodos anteriores criaram novos objetos sem modificar
#o objeto original (imutável)
print('Objetos imutáveis: ')
print(disciplina)
print(professor)

# Alguns operadores algebricos funcionam com Strings como:
# Para concatenar strings
print(disciplina + " em andamento")
# Ou para repetir uma string
print(5*"+++++")
# Acessando os caracteres da string
print("Código: ", disciplina[0:7])
# Por que utilizar aspas simples ou duplas?
stringComAspas = "Isto é uma 'String' que contem aspas simples"
print(stringComAspas)
stringComAspas = 'Agora a mesma "String" contem aspas duplas'
print(stringComAspas)
# As String são objetos inmutaveis então
try:
    disciplina[0] = professor[0]
except:
    print("Não posso modificar objetos inmutáveis!!!")
    print(disciplina)
```

```
Tamanho de disciplina:  34
Professor:  ESBEL T. VALERO ORELLANA
Disciplina:  cet1202 - algoritmos e programação
Alguem tem dúvidas?
Objetos imutáveis:
CET1202 - Algoritmos e Programação
Esbel T. Valero Orellana
CET1202 - Algoritmos e Programação em andamento
+++++++++++++++++++++++++
Código:  CET1202
```

```
Isto é uma 'String' que contem aspas simples
Agora a mesma "String" contem aspas duplas
Não posso modificar objetos inmutáveis!!!
CET1202 - Algoritmos e Programação
```

Tipo None O tipo **None** pode ser comparado, pela sua funcionalidade, ao tipo **NULL** do **C/C++**. Pode ser utilizado em diversos contextos mas, de forma geral, uma da suas aplicações mais frequentes é como o tipo de retorno padrão das funções. Veja o exemplo da função `print()`, lembrar que em **Python** todo pode ser entendido como referências a objetos.

[69]:
```
valor_de_retorno = print("Alguma coisa")
print(valor_de_retorno)
```

```
Alguma coisa
None
```

Tipo Boolean As variáveis de tipo *boolean* em **Python** podem conter dois valores possíveis, `True` ou `False`. Normalmente este é o tipo retornado pelos operadores relacionais. Variáveis *booleans* também podem ser construídas via *casting* explícito a partir de variáveis numéricas. Para os herdeiros de **C/C++**, qualquer valor numérico diferente de zero é convertido em `True`. *Strings* não vazias também geram `True`. As *Strings* vazias, a variável `None` e zero sempre são convertidos em `False`.

[70]:
```
x = 5.0
b = 3
txt = "texto"
zero = 0
nada = ""
nulo = None
print("x > 4.0 -> ", x > 4.0)
print("b == 4 -> ", b == 4)
print("bool(x) -> ", bool(x))
print("bool(zero) -> ", bool(zero))
print("bool(nada) -> ", bool(nada))
print("bool(nulo) -> ", bool(nulo))
```

```
x > 4.0 ->  True
b == 4 ->  False
bool(x) ->  True
bool(zero) ->  False
bool(nada) ->  False
bool(nulo) ->  False
```

Python: Operadores Após esclarecermos de forma resumida como se trabalha com variáveis em **Python**, e antes de revisar os diferentes tipos básicos disponíveis, vamos analisar os operadores que estão disponíveis para processar os dados armazenados nestas variáveis.

Operadores aritméticos Em **Python** são implementados um conjunto básico de operadores binários e unários, comumente utilizadas em outras linguagens.

Operador	Nome	Descrição
`a + b`	Adição	Soma de `a` e `b`
`a - b`	Subtração	Diferença entre `a` e `b`
`a * b`	Multiplicação	Producto de `a` e `b`
`a / b`	Divisão real	Quociente de `a` e `b`
`a // b`	Divisão truncada	Quociente de `a` e `b`, removendo a parte fracionária
`a % b`	Módulo	Resto inteiro da divisão de `a` por `b`
`a ** b`	Exponenciação	`a` elevado à potência de `b`
`-a`	Unário negativo	O negativo de `a`

Estes operadores pode ser utilizado de forma individual ou combinados e agrupados pelas regras de precedência, ou utilizando parênteses.

As regras de precedência dos operadores aritméticos estabelece que, em uma expressão que envolva diferentes operadores, serão avaliadas primeiramente as expressões dentro de parênteses e depois aquelas que utilizam o operador de exponenciação. Posteriormente são avaliadas aquelas que envolvem os operadores unários de sinal e somente depois as operações de multiplicação, divisão e resto da divisão, que possuem a mesma ordem de precedência. Operações de adição e subtração serão as últimas a serem avaliadas.

Quando uma expressão envolver mais de um operador com a mesma ordem de precedência, a mesma é avaliada da esquerda para direita.

Uma novidade, em relação a outras linguagens de programação, é a existência de dois operadores para a divisão. Nas primeiras versões de **Python** (**Python 2**) operador de divisão se comportava de acordo com as variáveis envolvidas na expressão:

- Quando a expressão envolve apenas variáveis inteiras se implementa a divisão truncada que retorna um valor inteiro;
- Quando a expressão envolve pelo menos uma variável de ponto flutuante se implementa a divisão real que retorna um valor fracionário;

O novo operador de divisão real, implementados a partir de **Python 3**, independe da natureza das variáveis numéricas envolvidas e sempre retornam um valor de ponto flutuante. Veja os exemplos a seguir?

[71]:
```
#Divisão real
print("1 / 3     -> ", 1 / 3)      # variáveis inteiras
print("6 / 3     -> ", 6 / 3)
print("256 / 10     -> ", 256 / 10)
print("1 / 3.0   -> ", 1 / 3.0)    # variáveis de ponto flutuante
print("6.0 / 3   -> " , 6.0 / 3)
print("256.0 / 0.5 -> ", 256.0 / 0.5)
```

```
1 / 3     ->  0.3333333333333333
6 / 3     ->  2.0
```

```
256 / 10     ->  25.6
1 / 3.0   ->  0.3333333333333333
6.0 / 3   ->  2.0
256.0 / 0.5 ->  512.0
```

Já o operador de divisão truncada preserva o comportamento do operador de divisão tradicional de outras linguagens. Veja os exemplos?

[72]:
```
#Divisão truncada
print("1 // 3     -> ", 1 // 3)      # variáveis inteiras
print("6 // 3     -> ", 6 // 3)
print("256 // 10     -> ", 256 // 10)
print("1 // 3.0   -> ", 1 // 3.0)    # variáveis de ponto flutuante
print("6.0 // 3   -> " , 6.0 // 3)
print("256.0 // 0.5 -> ", 256.0 // 0.5)
```

```
1 // 3     ->  0
6 // 3     ->  2
256 // 10     ->  25
1 // 3.0   ->  0.0
6.0 // 3   ->  2.0
256.0 // 0.5 ->  512.0
```

Operadores bit a bit Complementando os operadores aritméticos, **Python** define um conjunto de operadores que permitem trabalhar operações lógicas bit a bit em variáveis inteiras.

Operador	Nome	Descrição
`a & b`	AND bit a bit	Bits igualmente definidos em `a` e `b`
a | b	OR bit a bit	Bits definidos em `a` ou em `b` ou em ambos
`a ^ b`	XOR bit a bit	Bits definidos em `a` ou em `b` mas não em ambos
`a << b`	Deslocamento à esquerda	Desloca os bits de `a` à esquerda `b` unidades
`a >> b`	Deslocamento à direita	Desloca os bits de `a` à direita `b` unidades
`~a`	NOT bit a bit	Negação bit a bit de `a`

Os operadores bit a bit somente fazem sentido quando trabalhamos os valores inteiros utilizando sua representação em binário. Cada bit pode estar associado com algum tipo de informação específica (um interruptor ligado, um gene activado, ...). Neste caso podemos trabalhar os bits individualmente de forma bastante eficiente.

Imaginemos, por exemplo, que cada um dos bits de um inteiro representam genes ativados ou não numa determinada sequência genética.

[73]:
```
# Quais genes estão ativados no individuo com sequencia 120?
ind_1 = 120
bin(ind_1)
```

[73]: '0b1111000'

[74]:
```
#Quais genes estão ativados no individuo com sequencia 63?
ind_2 = 63
bin(ind_2)
```

[74]: '0b111111'

[75]:
```
# Quais genes estão ativados em ambos indivíduos?
print("     ", ind_1 , " -> ", bin(ind_1))
print("     ", ind_2 , "  ->  ", bin(ind_2))
print("AND: ", ind_1 & ind_2, "  ->  ", bin(ind_1 & ind_2))
print("_____________________________________")
# Quais genes estão ativados em um ou em outro indivíduo?
print("     ", ind_1 , " -> ", bin(120))
print("     ", ind_2 , "  ->  ", bin(ind_2))
print("OR : ", ind_1 | ind_2, " -> ", bin(ind_1 | ind_2))
print("_____________________________________")
# Quais genes estão ativados em um ou em outro indivíduo mas não em ambos?
print("     ", ind_1 , " -> ", bin(ind_1))
print("     ", ind_2 , "  ->  ", bin(ind_2))
print("XOR: ", ind_1 ^ ind_2, "  -> ", bin(ind_1 ^ ind_2))
# Garanta que o gene mais a direita do segundo indivíduo esteja desligado
print("_____________________________________")
print(ind_2, " -> ", bin(ind_2))
ind_2 = (ind_2 >> 1) << 1
print(ind_2, " -> ", bin(ind_2))
print("_____________________________________")
```

```
      120  ->  0b1111000
      63   ->   0b111111
AND:  56   ->   0b111000
-----------------------------------------
      120  ->  0b1111000
      63   ->   0b111111
OR :  127  ->  0b1111111
-----------------------------------------
      120  ->  0b1111000
      63   ->   0b111111
XOR:  71   ->  0b1000111
-----------------------------------------
63  ->  0b111111
62  ->  0b111110
-----------------------------------------
```

Operador de atribuição Até agora utilizamos o operador =, de forma natural, como operador de atribuição. Entretanto, nas variáveis imutáveis como as numéricas, no momento da atribuição não está se atualizando ou alterando o valor contido no espaço de memória referenciado pela variável,

como acontece em outras linguagens mais tradicionais. Trata-se de uma atualização do objeto que está sendo referenciado.

Quando utilizado em conjunto com os operadores binários, apresentados anteriormente, o operador de atribuição funciona como um operador composto. Ou seja: para cada operador binário `#`, é possível substituir a operação `a = a # b` por `a #= b`. Desta forma teremos

Operador	Exemplo	Equivalente
`+=`	`a += b`	`a = a + b`
`-=`	`a -= b`	`a = a - b`
`*=`	`a *= b`	`a = a * b`
`/=`	`a /= b`	`a = a / b`
`//=`	`a //= b`	`a = a //  b`
`%=`	`a %= b`	`a = a % b`
`**=`	`a **= b`	`a = a ** b`
`&=`	`a &= b`	`a = a & b`
`|=`	a |= b	a = a | b
`^=`	`a ^= b`	`a = a ^ b`
`<<=`	`a <<= b`	`a = a << b`
`>>=`	`a >>= b`	`a = a >> b`

Operadores lógicos e relacionais Para trabalhar expressões de comparação **Python** implementa um conjunto de operadores básicos que retornam `True` ou `False`.

Operação	Descrição	Operação	Descrição
`a == b`	`a` igual a `b`	`a != b`	`a` diferente de `b`
`a < b`	`a` menor que `b`	`a > b`	`a` maior que `b`
`a <= b`	`a` menor ou igual que `b`	`a >= b`	`a` maior ou igual que `b`

Veja alguns exemplos que combinam operadores aritméticos e relacionais:

[76]:
```
a = 6 / 5
b = 7 / 8
print( " 6 / 5 > 7 / 8 -> ", a > b)
print( " 6 / 5 < 7 / 8 -> ", a < b)
print( " 6 / 5 == 7 / 8 -> ", a == b)
print( " 6 / 5 != 7 / 8 -> ", a != b)
print( " 6 / 5 >= 7 / 8 -> ", a >= b)
print( " 6 / 5 <= 7 / 8 -> ", a <= b)
```

```
 6 / 5 > 7 / 8 ->  True
 6 / 5 < 7 / 8 ->  False
 6 / 5 == 7 / 8 ->  False
 6 / 5 != 7 / 8 ->  True
 6 / 5 >= 7 / 8 ->  True
 6 / 5 <= 7 / 8 ->  False
```

[77]:
```
print("6 é um número par? -> ", (6 % 2) == 0)
print(" e 7 é par? -> ", (7 % 2) == 0)
```

```
6 é um número par? ->  True
 e 7 é par? ->  False
```

Veja um exemplo um pouco mais complexo de uso de operadores relacionais.

[78]:
```
L = [ 10, 25, 40]
print("Valores de L dentro do intervalo fechado (15, 30)")
for x in L:
    print("15 < ", x, " < 30 -> ", 15 < x < 30)
```

```
Valores de L dentro do intervalo fechado (15, 30)
15 <  10  < 30 ->  False
15 <  25  < 30 ->  True
15 <  40  < 30 ->  False
```

As operações envolvendo variáveis com valores booleanos, muitas vezes precisam ser combinadas. Para trabalhar estas operações de álgebra booleana estão implementados os operadores básicos `and`, `or` e `not`. O operador `xor` lógico não está explicitamente definido na linguagem.

[79]:
```
print("Valores de L dentro do intervalo fechado (15, 30)")
for x in L:
    print("15 < ", x, " < 30 -> ", (15 < x) and (x < 30))
```

```
Valores de L dentro do intervalo fechado (15, 30)
15 <  10  < 30 ->  False
15 <  25  < 30 ->  True
15 <  40  < 30 ->  False
```

[80]:
```
print("Valores de L fora do intervalo fechado (15, 30)")
for x in L:
    print("15 >= ", x, " >= 30 -> ", (15 >= x) or (x >= 30))
```

```
Valores de L fora do intervalo fechado (15, 30)
15 >=  10  >= 30 ->  True
15 >=  25  >= 30 ->  False
15 >=  40  >= 30 ->  True
```

Operadores identidade e pertence **Python** introduz também dois operadores particularmente importantes que serão muito utilizados. Estes operadores permitam avaliar se duas referências estão se apontando ao mesmo objeto (operador identidade) e se u um objeto pertence a um conjunto.

Operador	Descrição
`a is b`	`True` se `a` e `b` são objetos idênticos
`a is not b`	`True` se `a` e `b` não são objetos idênticos
`a in b`	`True` se `a` é membro do conjunto `b`

Operador	Descrição
`a not in b`	True se `a` não é membro do conjunto `b`

Importante ressaltar que objetos idênticos não é a mesma coisa que objetos iguais. Veja o seguinte exemplo.

[81]:
```
Lc = [10, 25, 40]
print("L: ", L)
print("Lc: ", Lc)
print("1 - L é igual a Lc?", L == Lc)
print("2 - L é idêntico a Lc?", L is Lc)
Lc = L
print("3 - L é igual a Lc?", L == Lc)
print("4 - L é idêntico a Lc?", L is Lc)
```

```
L:  [10, 25, 40]
Lc:  [10, 25, 40]
1 - L é igual a Lc? True
2 - L é idêntico a Lc? False
3 - L é igual a Lc? True
4 - L é idêntico a Lc? True
```

O operador `in` é um exemplo clássico dos recursos que fazem de **Python** um linguagem de fácil leitura e escrita. As operações de busca em conjuntos são, de forma geral, implementadas utilizando estruturas de repetição custosas e, geralmente, de difícil leitura. Veja estes exemplos simples, que funcionam de forma eficiente.

[82]:
```
print("L: ", L)
print("5 pertence a L?", 5 in L)
print("5 não pertence a L?", 5 not in L)
print("25 pertence a L?", 25 in L)
print("37 não pertence a L?", 37 not in L)
```

```
L:  [10, 25, 40]
5 pertence a L? False
5 não pertence a L? True
25 pertence a L? True
37 não pertence a L? True
```

Para finalizar veja a tabela a seguir que apresenta os operadores em ordem decrescente de precedência em **Python**

Operador	Descrição
`()`	Parênteses
`**`	Exponenciação
`+x`, `-x`, `~x`	Unário positivo, unário negativo, NOT bit a bit
`*`, `/`, `//`, `%`	Multiplicação, Divisão real, Divisão truncada, Módulo
`+`, `-`	Adição, Subtração

Operador	Descrição
`<<, >>`	Deslocamentos bit a bit
`&`	Operador `AND` bit a bit
`^`	Operador `XOR` bit a bit
`|`	Operador `OR` bit a bot
`==, !=, >, >=, <, <=, is, is not, in, not in`	Operadores relacionais, identidade e pertence
`not`	Operador `NOT` lógico
`and`	Operador AND`lógico | |or| Operador ÒR`lógico

Tipos de dados estruturados nativos Além dos tipos mais simples **Python** fornece um conjunto de tipos de dados estruturados muito rico e interessante.

Nome	Exemplo	Descrição
`list`	`[1, 2, 3]`	Coleção ordenada
`tuple`	`(1, 2, 3)`	Coleção imutável ordenada
`dict`	`{'a':1, 'b':2, 'c':3}`	Mapeamento do tipo (chave, valor)
`set`	`{1, 2, 3}`	Coleção não ordenada de valores únicos

Novamente vamos nos debruçar sobre cada um destes tipos.

Listas Em **Python** uma lista é definida como uma estrutura básica, ordenada e mutável. Mais espec[ificamente, uma lista se define como um conjunto de valores, separados por virgula e delimitados por `[ ... ]`. Veja o seguinte exemplo, onde demonstramos algumas das propriedades e métodos das listas. Maiores informações sobre listas podem ser acessadas na documentação oficial More on Lists

[83]:
```
# A lista dos 5 primeiros números pares
P = [ 2, 4, 6, 8, 10]
print("Uma lista P formada por números pares: ", P)
# Tamanho da lista: a função len() retorna o tamanho da lista
print("De tamanho len(P): ", len(P))
#Acrescentando um valor no final da lista
print("Adicionando o elemento 12 no final da lista.")
P.append(12)
print("A lista agora ficou assim: ", P)
# Os operadores aritméticos também funcionam com listas da mesma forma que
# com strings
# Concatenando lista
print("Uma nova lista PpI resultado de concatenar P com uma lista de números␣
 ↪ímpares")
PpI = P + [1, 3, 5, 7, 9, 11]
print(PpI)
# Repetindo os elementos da lista
print("A lista Pt3 resultado de repetir os elementos de P 3 vezes")
```

```
Pt3 = 3*P
print(Pt3)
# Ordenando a lista
PpI.sort()
print("A lista PpI ordenada com o método short(): ", PpI)
# A lista, ao contrario de uma string, é mutável
print("A lista ordenada: ", PpI)
PpI[0] = 0
print("Alterando o elemento PpI[0]: ", PpI)
```

```
Uma lista P formada por números pares:  [2, 4, 6, 8, 10]
De tamanho len(P):  5
Adicionando o elemento 12 no final da lista.
A lista agora ficou assim:  [2, 4, 6, 8, 10, 12]
Uma nova lista PpI resultado de concatenar P com uma lista de números ímpares
[2, 4, 6, 8, 10, 12, 1, 3, 5, 7, 9, 11]
A lista Pt3 resultado de repetir os elementos de P 3 vezes
[2, 4, 6, 8, 10, 12, 2, 4, 6, 8, 10, 12, 2, 4, 6, 8, 10, 12]
A lista PpI ordenada com o método short():  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
12]
A lista ordenada:  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
Alterando o elemento PpI[0]:  [0, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

Mas se você está achando que as listas em **Python** são o equivalente aos *arrays* em outras linguagens, como **C/C++**, está errado. As listas que vimos até agora armazenam dados do mesmo tipo. Mas as listas estão mais para listas de compras onde pode entrar líquidos, sólidos, perecíveis, biscoitos, legumes, cerveja, itens de limpeza, por quilos, por litros, por quantidade, etc. Ou seja, listas não se restringem a um tipo específico, e podem armazenar dados de tipos diferentes. Uma lista pode até ser formada por outras listas. Veja os exemplos a seguir.

[84]:
```
Lh = [1, 2.0, "três", [16//4, 10/2]]
print(Lh)
try:
    print(Lh.sort())
except Exception as inst:
    print("Não consegui ordenar: ", inst)
#print(Lh.sort())
```

```
[1, 2.0, 'três', [4, 5.0]]
Não consegui ordenar:  '<' not supported between instances of 'str' and 'float'
```

Indexando listas, slicing Os elementos de uma lista podem ser acessados individualmente através do índice que identifica cada um de maneira única. Em **Python**, da mesma forma que em **C/C++**, se implementa a indexação começando em zero. Como novidade, podemos utilizar índices negativos para acessar a lista de trás para frente. Desta forma:

[85]:
```
# A lista PpI anteriormente definida
print("Lista: ", PpI)
```

```
# O primeiro elemento da lista é
print("PpI[0] = ", PpI[0])
# e o segundo é
print("PpI[1] = ", PpI[1])
# O tamanho da lista
n = len(PpI)
print("Tamanho da lista n = ", n)
# O último elemento da lista pode ser acessado como:
print("PpI[n-1] = ", PpI[n-1])
# O é então como:
print("PpI[-1] = ", PpI[-1])
# e o anterior a ele
print("PpI[-2] = ", PpI[-2])
print("Utilizando indexação fora dos limites da lista")
try:
    print("O elemento PpI[n] = ", PpI[n])
except Exception as inst:
    print("Não consegui imprimir: ", inst)
```

```
Lista:  [0, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
PpI[0] =  0
PpI[1] =  2
Tamanho da lista n =  12
PpI[n-1] =  12
PpI[-1] =  12
PpI[-2] =  11
Utilizando indexação fora dos limites da lista
Não consegui imprimir:  list index out of range
```

Os elementos podem ser acessados também como sub listas geradas utilizando o mecanismo de *slicing*. O *slicing* consiste do uso de uma sintaxe, baseada no emprego de `[ini:fim]`, para identificar o início (`ini`) da sub lista, incluído nela, e o final, não incluído. Podemos omitir o início da sub lista (`[:fim]`) e, neste caso, a mesma começa no elemento de índice 0. Também podemos omitir o final (`[ini:]`) e serão incluídos os elementos até o final da lista. Um terceiro inteiro pode ser utilizado no slicing, `[ini:fim:passo]` para definir o intervalo entre um elemento e outro a ser selecionado para integrar a sub lista. Veja os exemplos a seguir:

[86]:
```
# Os primeiros três elementos de P
print("P[:3] = ", P[:3]) # equivale a P[0:3], retorna os elemento de índice 0,␣
 ↪1 e 2
# Os elementos da lista excluindo o primeiro
print("P[1:] = ",P[1:]) # equivale a P[1:len(P)]
# Uma sub lista da original, retorna os elementos de índice 1, 2 e 3
print("P[1:4] = ", P[1:4])
# Extraindo os elementos com índice par da lista completa
print("PpI[::2] = ",PpI[::2]) # equivale a PpI[0:len(PpI):2]
```

```
P[:3] =  [2, 4, 6]
P[1:] =  [4, 6, 8, 10, 12]
```

```
P[1:4] =  [4, 6, 8]
PpI[::2] =  [0, 3, 5, 7, 9, 11]
```

A sintaxes do *slicing* pode ser utilizada também em outras estruturas, como *strings*, tuplas ou nos arrays definidos em pacotes como o **NumPy** e o **Pandas**. Vejamos como mostrar uma string ao contrario usando slicing

[87]:
```
texto = "abcdefghijk"
print(texto[-1::-1])
```

```
kjihgfedcba
```

#####Tuplas

Tuplas são bastante similares a listas em alguns aspectos. São definidas como um conjunto de elementos separados por vírgula e delimitados por parênteses. As tuplas também tem tamanho definido (`len`) e os elementos da mesma também podem ser acessados pelo seu índice. A principal diferença é que as tuplas são imutáveis, ou seja, uma vez criadas seu tamanho e seu conteúdo não podem ser alterados. Veja os exemplos:

[88]:
```
dupla = (2,3.6)
print(dupla)
c = 3
tripla = ("a", 'b', c)
print(tripla)
print("tripla[0] -> ", tripla[0])
# Sobre o fato de serem imutaveis
try:
    tripla[2] = 3
except Exception as inst:
    print("Não consegui modificar a tupla: ", inst)
```

```
(2, 3.6)
('a', 'b', 3)
tripla[0] ->  a
Não consegui modificar a tupla:  'tuple' object does not support item assignment
```

As tuplas podem ser utilizadas em diversos contextos. Um exemplo simples pode ser o caso das funções que precisam retornar mais de uma variável. Veja este exemplo interessante que demonstra as potencialidade de **Python**.

[89]:
```
# X armazena um valor de ponto flutuante
x = 0.125
# Na realidade este valor pode ser representado como uma frção
fracão = x.as_integer_ratio()
print(type(fracão))
print(x.as_integer_ratio()) # método da classe dos objetos da classe ponto␣
 ↪flutuante
numerador, denominador = x.as_integer_ratio() # o método retorna uma dupla
print(numerador, "/", denominador) # aqui escrevendo como uma fração
```

```
print(numerador / denominador) # aqui escrevendo novamente como um ponto␣
  ↪flutuante
# referenciando Tuplas
p1 = (1.0, 2.3)
print("p1 = ", p1)
p2 = p1
print("p2 = ", p2)
p1 = (1.0, 2.3, 3.4)
print("p1 = ", p1)
print("p2 = ", p2)
# Sobre indexamento e tamanho das tuplas
print("P1 é uma tupla de ", len(p1), " componentes.")
print("O último componente da tupla é P1[-1] = ", p1[-1])
```

```
<class 'tuple'>
(1, 8)
1 / 8
0.125
p1 =  (1.0, 2.3)
p2 =  (1.0, 2.3)
p1 =  (1.0, 2.3, 3.4)
p2 =  (1.0, 2.3)
P1 é uma tupla de  3  componentes.
O último componente da tupla é P1[-1] =  3.4
```

Veja este outro exemplo utilizando o classico problema de trocar os valores de duas variáveis:

[90]:
```
a = 5
b = 4
print("a = ", a, " e b = ", b)
# Trocando da forma tradicional
swap = a
a = b
b = swap
print("a = ", a, " e b = ", b)
#trocando usando tuplas
a, b = b, a
print("a = ", a, " e b = ", b)
```

```
a =  5  e b =  4
a =  4  e b =  5
a =  5  e b =  4
```