

Programação Científica

Prof. Dr. Danilo H. Perico

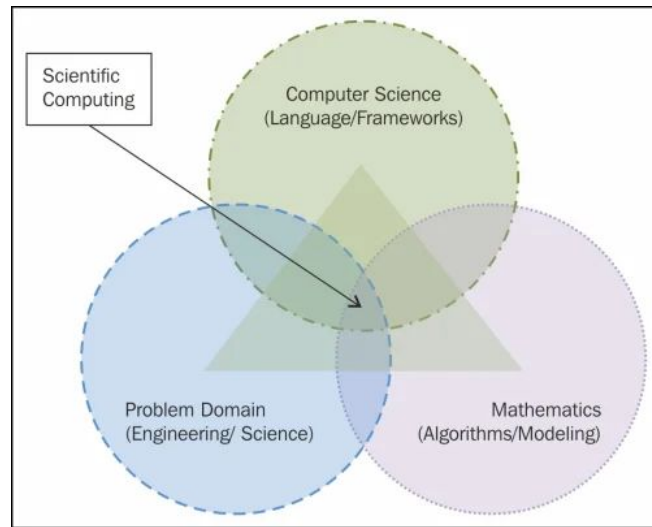
Programação Científica

- Nomes alternativos na literatura: ***Scientific Programming*** ou ***Scientific Computing***

O nome é usado como termo genérico para ferramentas computacionais sendo utilizadas para a solução de problemas científicos ou de engenharia

Programação Científica

- Programação Científica é o título dado a um campo **interdisciplinar**: envolve Ciência da Computação, Modelagem do Problema (Ciência e/ou Engenharia) e Matemática



Fonte imagem:

<https://subscription.packtpub.com/book/big-data-and-business-intelligence/9781783288823/1/ch01lv1sec08/definition-of-scientific-computing>

Programação Científica

“A computação científica é a coleção de ferramentas, técnicas e teorias necessárias para resolver em um computador os modelos matemáticos de problemas em ciência e engenharia.”

Gene H. Golub e James M. Ortega

*(Scientific Computing and Differential Equations: An
Introduction to Numerical Methods)*

Programação Científica

- O livro ***A Primer on Scientific Programming with Python*** (Hans Petter Langtangen) especifica que seu principal objetivo é ensinar programação com exemplos na matemática e nas ciências naturais
- O autor **Robert Johansson** (*Numerical Python*) define:

“Scientific Computing is often viewed as a new branch of science. In most fields of science, computational work is an important complement to both experiments and theory, and nowadays a vast majority of both experimental and theoretical papers involve some numerical calculations, simulations or computer modeling”

Programação Científica

- “Programação Científica é não linear e exploratória”

Jake VanderPlas (*Python Data Science Handbook*)

Programação Científica

- **Robert Johansson** fala ainda sobre o importante papel da Programação Científica no método científico:
 - “**Replication** and **reproducibility** are two of the cornerstones in the scientific method”

Programação Científica

- **Replication:** *An author of a scientific paper that involves numerical calculations should be able to rerun the simulations and replicate the results upon request. Other scientist should also be able to perform the same calculations and obtain the same results, given the information about the methods used in a publication.*
- **Reproducibility:** *The results obtained from numerical simulations should be reproducible with an independent implementation of the method, or using a different method altogether*

Programação Científica

- De maneira resumida:
 - Um resultado científico sólido deve ser **reprodutível** e um estudo científico sólido deve ser **replicável**

Programação Científica

- Para atingir esses objetivos, vamos trabalhar com a **Programação Científica!**
 - Lembre-se que, idealmente, os códigos utilizados em experimentos devem ser publicados online para facilitar o acesso de sua pesquisa para outros cientistas!
 - Se o código for construído com padrões conhecidos e com o auxílio de bibliotecas e ferramentas bem estabelecidas, a **replicabilidade** será facilmente alcançada e o método proposto terá resultados **reprodutíveis**

Programação Científica

Objetivo da disciplina:

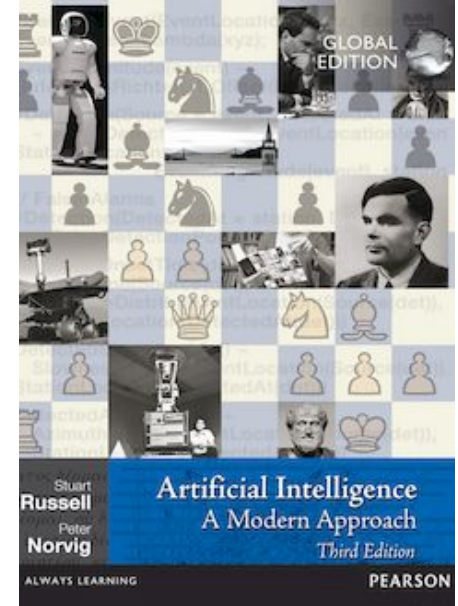
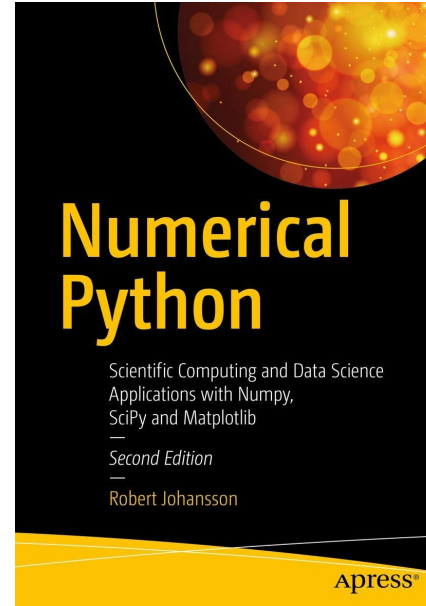
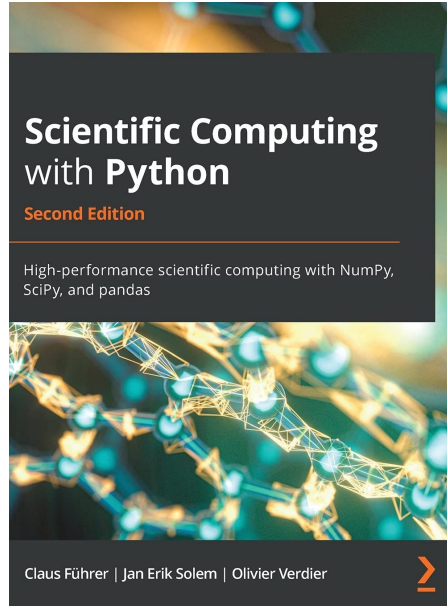
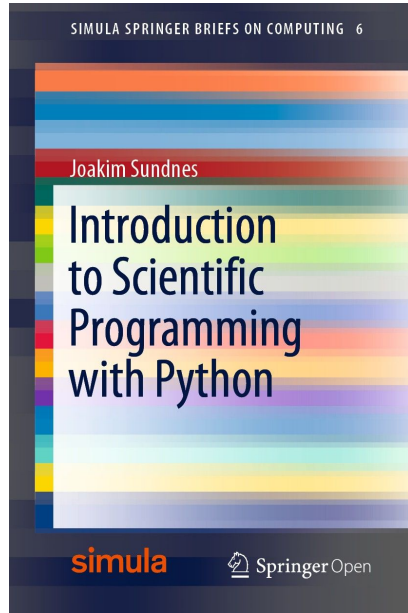
- Apresentar aos alunos de Pós-Graduação os conceitos associados a programação científica, bem como algumas das principais ferramentas para o desenvolvimento de computação numérica, simbólica e paralela. Apresentar também ferramentas para a realização de tratamento, análise e visualização de dados. Aplicar os conceitos da programação científica na solução de problemas reais com o auxílio de técnicas de Otimização e de Inteligência Artificial.

Programação Científica

Vamos usar **Python** !

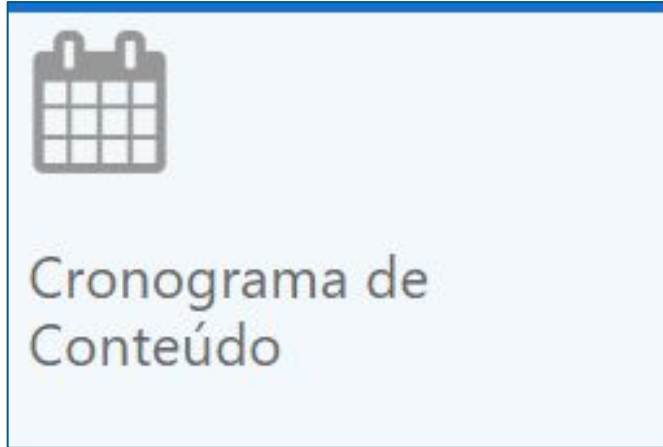


Bibliografia Básica



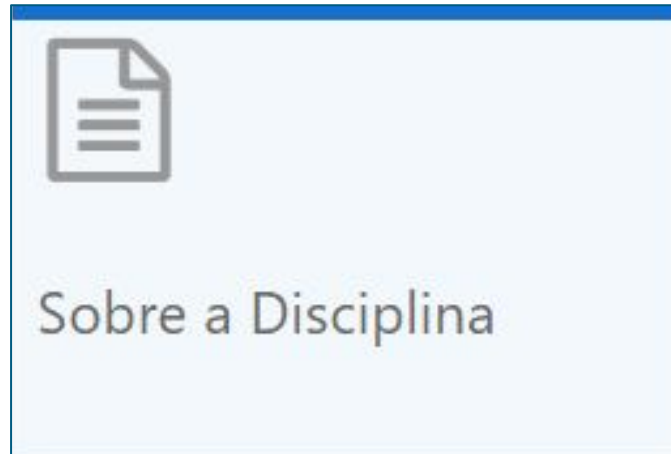
Conteúdo Programático

- No Moodle:
 - **Cronograma de Conteúdo**



Critério de Avaliação

- No Moodle:
 - **Sobre a Disciplina**





Python

Python

- É uma **Linguagem de Programação!**
- Linguagens de programação são usadas como um meio de comunicação entre os computadores e os humanos
- Codificam os algoritmos para uma linguagem que o computador pode entender
- Língua de alto nível
- Interpretada

Python - Breve História

- A Linguagem Python foi concebida no fim dos anos 80.
- A primeira ideia de implementar o Python surgiu mais especificamente em 1982 por **Guido Van Rossum**
- 1991: lançada a primeira versão do Python, então denominada de v0.9.0.



Por que usar Python???

- **Vantagens gerais:**

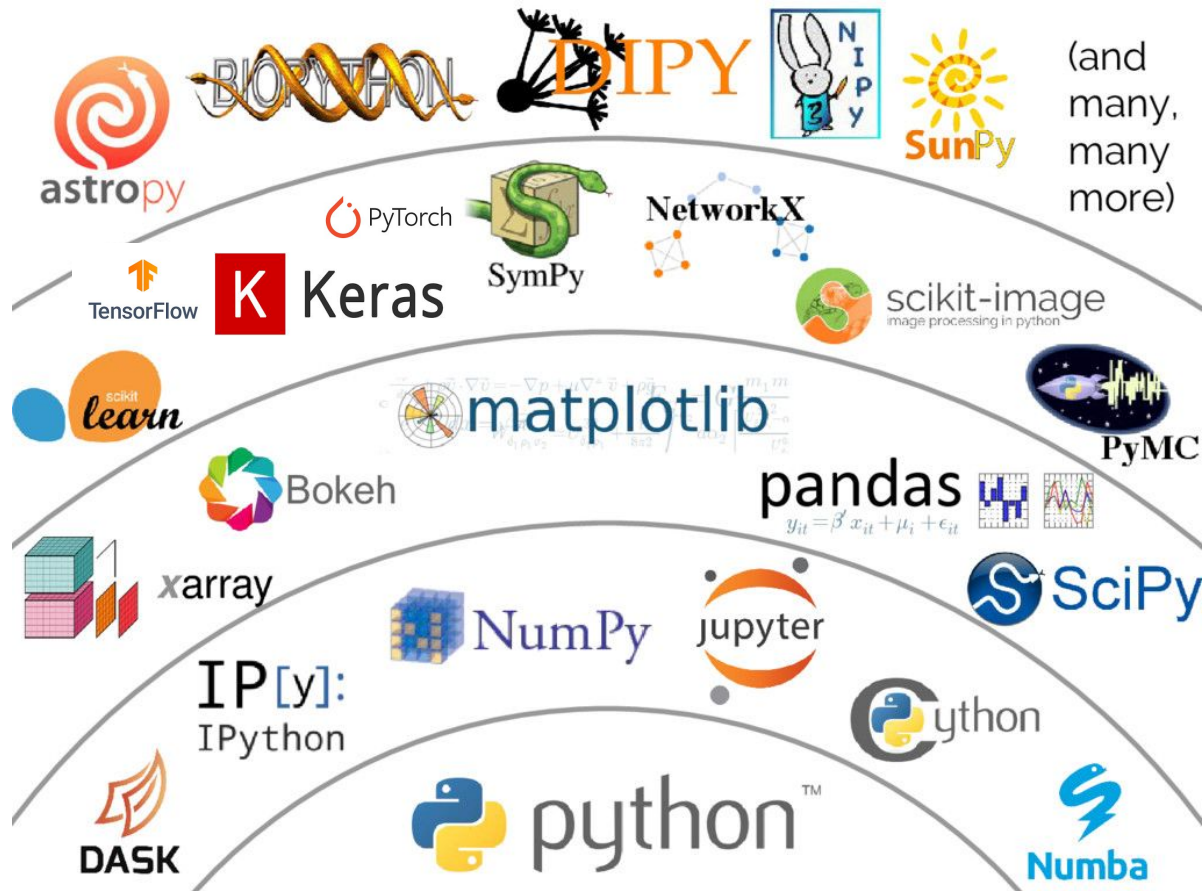
- **Fácil de programar** e aprender a programar
- É **portável** a quase todos os sistemas operacionais
- **Rápida prototipagem**
- Pode fazer **integração** com outras **linguagens**
- **Alta produtividade**

Por que usar Python???

- **Vantagens para Programação Científica:**
 - **Grande comunidade** de usuários
 - Amplo ecossistema de **bibliotecas** e **ambientes científicos**
 - Tem bom desempenho nas bibliotecas **escritas** em **C** e **Fortran**
 - Bom suporte para **processamento paralelo**, **computação em GPU** e **comunicação entre processos**
 - Adequado para uso em **clusters** de **computação de alto desempenho**
 - **Sem custos** de licença

Python for Scientific Computing

Alguns pacotes da pilha científica do Python



Python for Scientific Computing

“Python é cola!”

“Python junta e cola essa miscelânea de ferramentas científicas

Sua linguagem de alto nível pode empacotar bibliotecas feitas em C/FORTRAN de baixo nível *(linguagens responsáveis por fazer a computação acontecer no fim das contas)*”

Jake VanderPlas, PyCon 2017

Por que usar Python???

- **Desvantagem principal:**

- A execução do código em Python pode ser lenta em comparação com linguagens de programação compiladas estaticamente tipadas, como C e Fortran

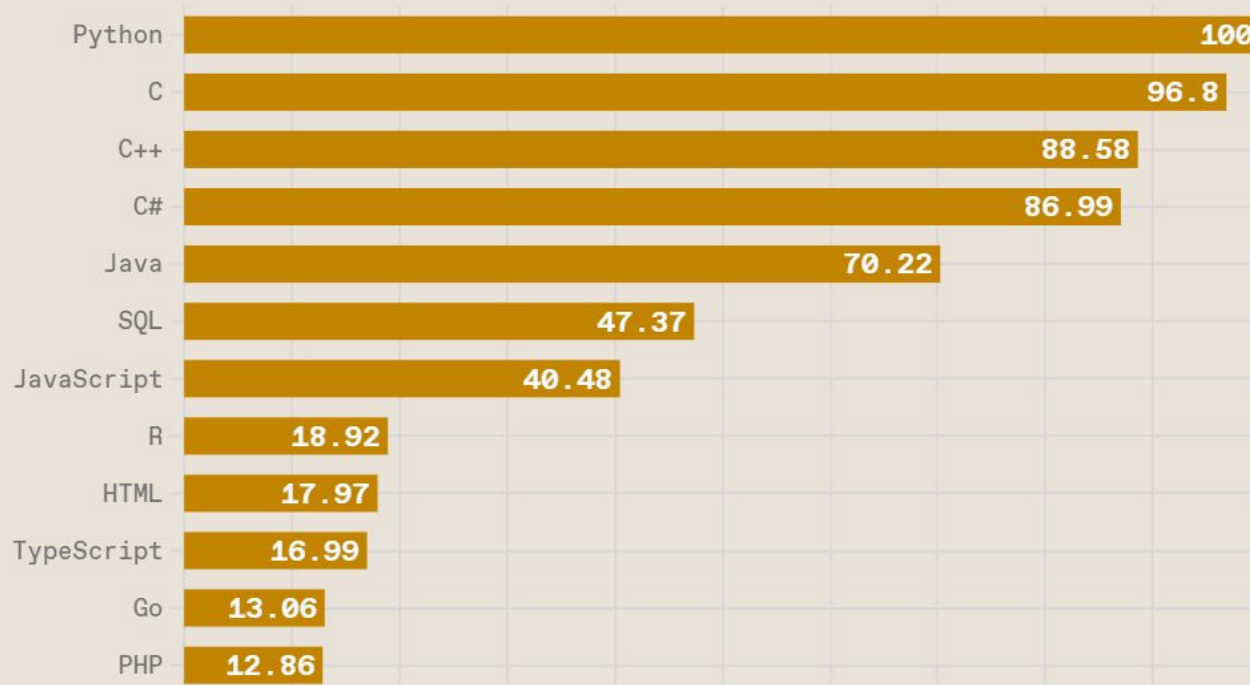
Top Programming Languages 2022

Click a button to see a differently weighted ranking

Spectrum











Jobs

Trending



Fonte:

<https://spectrum.ieee.org/top-programming-languages/>

Jul 2022	Jul 2021	Change	Programming Language		Ratings	Change
1	3	▲		Python	13.44%	+2.48%
2	1	▼		C	13.13%	+1.50%
3	2	▼		Java	11.59%	+0.40%
4	4			C++	10.00%	+1.98%
5	5			C#	5.65%	+0.82%
6	6			Visual Basic	4.97%	+0.47%
7	7			JavaScript	1.78%	-0.93%
8	9	▲		Assembly language	1.65%	-0.76%
9	10	▲		SQL	1.64%	+0.11%
10	16	▲		Swift	1.27%	+0.20%

Como programar em Python?

- Você pode baixar o programa Python no site:

<https://www.python.org/downloads/>

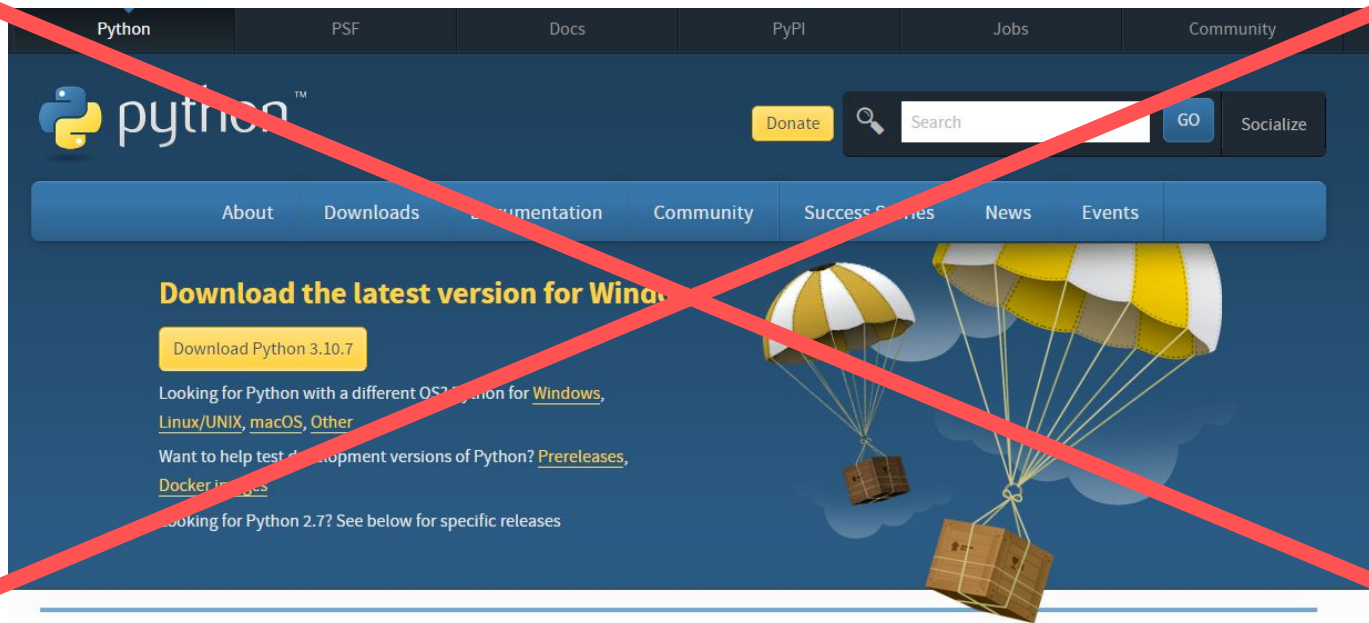


Como programar em Python?

Não recomendado!

- Você pode baixar o programa Python no site:

<https://www.python.org/downloads/>



Como programar em Python?

- Ou baixar o Python já com vários pacotes no site:

<https://www.anaconda.com/download/>



Products ▾

Pricing

Solutions ▾

Resources ▾

Partners ▾

Blog

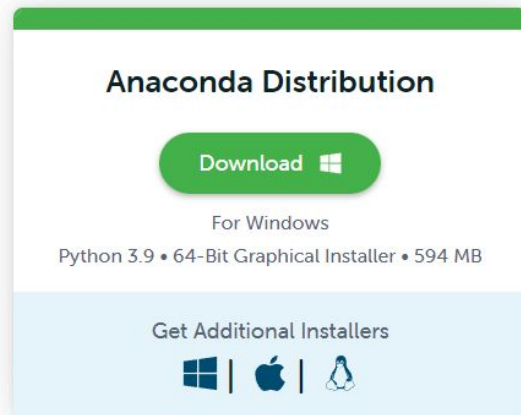
Company ▾

Contact Sales

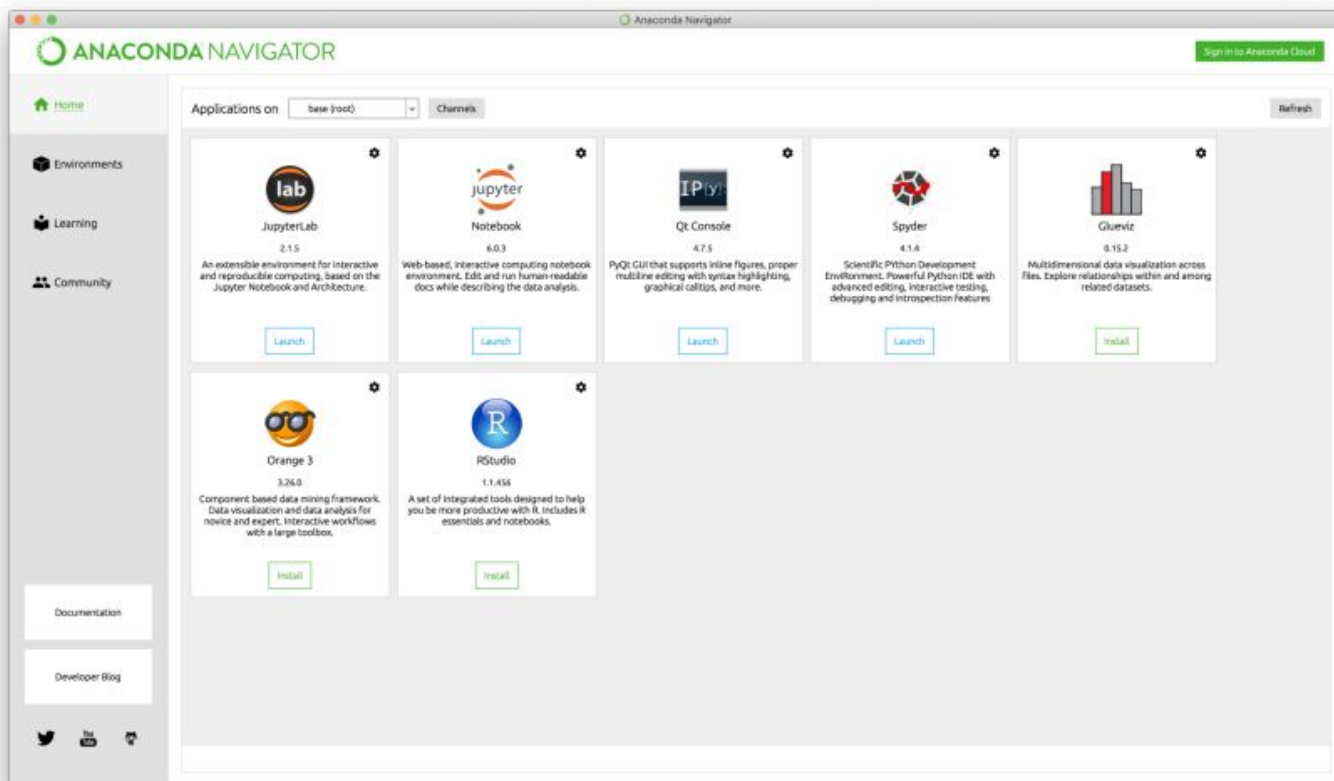
Individual Edition is now

ANACONDA DISTRIBUTION

The world's most popular open-source Python distribution platform



Navegador Anaconda





- Anaconda é uma distribuição das linguagens de programação **Python** e **R** para **computação científica** (*ciência de dados, Inteligência Artificial - incluindo Aprendizagem de Máquina, processamento de dados em larga escala, análise preditiva, etc.*)
- Visa simplificar o **gerenciamento** e a **implantação de pacotes**
- A distribuição inclui pacotes de **ciência de dados** adequados para Windows, Linux e macOS



- Vamos usar o *JupyterLab* ou o *Jupyter Notebook*



JupyterLab



Notebook

Python - Visão Geral da Linguagem

Variáveis

Variáveis

São espaços na memória no qual reservamos e damos nomes

Variáveis - Tipos de Dados

- **Dados** são as entidades mais fundamentais que um programa manipula!
- Os dados podem ser de diferentes **tipos**

Variáveis - Tipos de Dados

- Números:
 - Inteiro (*int*): 1 ; 2 ; -3 ; 0 ; 10
 - Real (*float*): 1.3 ; -3.63 ; 7.2 ; 16.42
 - Complexo (*complex*): $6 + 3j$; $-2 + 4j$
- Texto (*string*): “Olá” ; “Isto é uma string”
- Tipo Lógico (*bool*): True ; False

Variáveis - Nomes

- No Python os nomes das variáveis começam, obrigatoriamente, com uma **letra**
- Porém, o nome completo da variável pode conter números e o símbolo sublinhado (_)
- Em programação, normalmente **não se utiliza acentos** nos nomes das variáveis, porém o Python 3 permite esta utilização
- O Python diferencia letras maiúsculas de letras minúsculas

Variáveis - Atribuição de Valores

- Para armazenar valores nas variáveis, utilizamos o símbolo igual (=)
- Esta operação é chamada de **atribuição**, pois um valor é atribuído a uma variável
- Exemplo:

```
a = 5
b = 6
print(a + b)

11
```

- **a** recebe o valor 5
- **b** recebe o valor 6
- Imprimi o resultado da soma de **a** com **b**

Variáveis - Conversão de Valores

- A conversão de valores é feita por meio das funções:

- `int()`
- `float()`
- `str()`

- Exemplos:

```
In [9]: a = "3"  
        b = int(a)  
        type(b)
```

```
Out[9]: int
```

```
In [10]: a = 3  
         b = str(a)  
         type(b)
```

```
Out[10]: str
```

Entrada e Saída

Função *print()*

- Podemos imprimir *strings* ou o valor de variáveis com os seguintes comandos:

```
print("Olá mundo!")
```

```
Olá mundo!
```

```
a = 4  
print(a)
```

```
4
```

Função *print()*

- Para a impressão de dados na tela, podemos também combinar texto e o valor de alguma variável, utilizando vírgula (,) entre um dado e outro:

```
a = 3  
print("a vale", a)  
a vale 3
```

Função *print()* - composição

```
anos = 30
print("João tem %d anos" % anos)

João tem 30 anos
```

- O símbolo *%d* é chamado de **marcador**.
- O marcador indica que naquela posição estaremos colocando o valor da variável *anos*, que deve ser um número inteiro neste exemplo.
- Marcadores mais comuns:


marcador	tipo
%d	Número inteiro (int)
%f	Números decimais (float)
%s	Strings

Função *print()* - *format()*

- O método *format()* formata a *string* dada de acordo com o desejado para a saída de dados
 - Exemplo:

```
1 name = "Fulano"
2
3 peso = 78.51
4
5 print("0 {0} pesa {1:2.1f} Kg".format(name,peso))
```

0 Fulano pesa 78.5 Kg

A diagram with two green curved arrows. The first arrow starts from the variable 'name' in line 1 and points to the '{0}' placeholder in line 5. The second arrow starts from the variable 'peso' in line 3 and points to the '{1:2.1f}' placeholder in line 5.

Função *input()*

- A entrada de dados é feita pela função *input()*
- *input()* aceita como parâmetro uma mensagem a ser exibida
- O valor recebido pela entrada de dados deve ser atribuído a uma variável
- Todo valor recebido pela função *input()* tem sempre o tipo *string*
- Se a ideia é utilizar o valor recebido em alguma conta ou cálculo, ele deve ser convertido para algum tipo numérico

Função *input()*

- Exemplo:
 - O valor digitado para *pi* será recebido como string; antes de ser atribuído a variável *pi*, ele é convertido em float:

```
pi = float(input("Digite o valor de pi: "))  
print("O valor digitado é %.1f" % pi)
```

```
Digite o valor de pi: 3.14159  
O valor digitado é 3.1
```

Comentários

Comentários

- São textos que não são interpretados como código de programa
- Servem para documentar o programa
- No Python, uma linha de comentário começa com o símbolo # (padrão mais comum)
- Exemplo:

```
a = 8
#isso é um comentário
print(a)
```

8

Funções Matemáticas

Algumas das funções matemáticas mais usadas

- É necessário importar o módulo de matemática
- *from math import **

abs(x)	Valor absoluto de x: $ x $
sqrt(x)	Raiz quadrada de x
log(x)	Retorna o logaritmo natural de x: $\ln(x)$
log10(x)	Retorna o logaritmo base-10 de x
sin(x)	Retorna o seno de x radianos
cos(x)	Retorna o cosseno de x radianos
exp(x)	Retorna e^x
round(x, n)	Número x arredondado para n dígitos

Exemplo

```
from math import *  
  
print(sqrt(9))  
print(log(3))
```

3.0

1.0986122886681098

Estrutura de Seleção

Comando **if**

- Em Python, e em várias outras linguagens de programação, o comando principal para a realização de decisões é o **if**
- Sintaxe do **if** no Python:

```
if <condição>:  
    bloco verdadeiro
```

- **If** nada mais é do que nosso **se**
- Em português, podemos entender o comando if da seguinte forma:
 - **Se a condição for verdadeira, faça alguma coisa**

Comando **if** - Exemplo

Ler dois valores e apresentar o maior deles:

```
a = int(input("Primeiro Valor: "))  
b = int(input("Segundo Valor: "))  
  
if a > b:  
    print("O primeiro é o maior!")  
  
if b > a:  
    print("O segundo é o maior!")
```

```
Primeiro Valor: 87  
Segundo Valor: 54  
O primeiro é o maior!
```

Comando **if** - indentação

- O bloco que será executado se a condição do *if* for verdadeira fica **indentado** com relação ao comando *if*
- **Indentação** é o **recuo** (deslocamento do texto à direita)
- **Indentação**: neologismo derivado da palavra em inglês *indentation*
- ***Blocos são definidos pela indentação***

Comando **if** - indentação

```
if a > b:  
    print("O primeiro é o maior!")  
    print(a)  
    print("fim do if")
```

↑
indentação

Comando **if** - indentação - Exemplo

```
a = int(input("Primeiro Valor: "))  
b = int(input("Segundo Valor: "))
```

```
if a > b:
```

```
    print("O primeiro é o maior!")  
    print(a)  
    print("fim do if")
```



Estes são os comandos que pertencem ao bloco do if

```
print("Este print() executa de forma independente com relação à condição a > b")
```

Comando **else**

- O comando **else** (senão) é utilizado nos casos em que a segunda condição é simplesmente o **contrário** da primeira
- Sempre utilizado como uma sequência de um **if**
- Sintaxe:

```
if <condição>:  
    bloco verdadeiro  
else:  
    bloco contrário
```

Comando **else**

```
a = int(input("Digite o primeiro valor: "))
b = int(input("Digite o segundo valor: "))

if a > b:
    print("O primeiro é maior")

if a < b:
    print("O segundo é maior")
else:
    print("Os números são iguais")
```

```
Digite o primeiro valor: 10
Digite o segundo valor: 10
Os números são iguais
```

Comando **elif**

- O comando **elif** (*else if* - senão se) substitui, em muitos casos, a necessidade do aninhamento
- É sempre utilizado como uma sequência de um **if**
- Sintaxe:

```
if <condição 1>:  
    #bloco se a condição 1 for verdadeira  
elif <condição 2>:  
    #bloco se a condição 1 for falsa e a condição 2 verdadeira  
else:  
    #bloco contrário a todas outras condições
```

Comando **elif** - Exemplo

- Exemplo da conta de telefone alterado para usar **elif**

```
minutos = int(input("Quantos minutos foram utilizados este mês: "))

if minutos < 200:
    preco = 0.20
elif minutos < 400:
    preco = 0.18
else:
    preco = 0.15

print("O valor da sua conta é R$ %.2f" % (minutos*preco))
```

```
Quantos minutos foram utilizados este mês: 485
O valor da sua conta é R$ 72.75
```

inline if-else expression

inline if-else expression

- Podemos criar estruturas de seleção *inline*
- Sintaxe:

`<on_true> if <expression> else <on_false>`

- Exemplo:

```
idade = 20
print("Criança" if idade < 18 else "Adulto")

Adulto
```

inline if-else expression

- Exemplo:

```
numero = 10  
par = True if numero%2==0 else False  
print(par)
```

True

Operadores Lógicos

Operadores Lógicos

- Podemos **combinar condições** para determinar como continuar o fluxo de um programa!
- Os operadores lógicos mais utilizados são:
 - ***and (E)***
 - ***or (OU)***
 - ***not (NÃO)***

Operadores Lógicos

Lembre-se sempre:

- **Operadores relacionais retornam sempre um valor Booleano:**
 - **True** ou **False**

Operadores Lógicos - *and*

- Operador *and*:
 - Também retorna **True** ou **False** na comparação das condições
 - Todas as condições devem ser verdadeiras para o *and* retornar **True**

and (e)	Comparação 1	Comparação 2	Resultado
	True	True	True
	True	False	False
	False	True	False
	False	False	False

Operadores Lógicos - **or**

- Operador **or** :
 - Também retorna **True** ou **False** na comparação das condições
 - Basta que uma condição seja verdadeira para o **or** retornar **True**

or (ou)	Comparação 1	Comparação 2	Resultado
	True	True	True
	True	False	True
	False	True	True
	False	False	False

Exercício 1

Faça um programa que faça 5 perguntas para uma pessoa sobre um crime. As perguntas são:

- “Telefonou para a vítima?”
- “Esteve no local do crime?”
- “Mora perto da vítima?”
- “Devia para a vítima?”
- “Já trabalhou com a vítima?”

Então, o programa deve emitir uma classificação sobre a participação da pessoa no crime. Se a pessoa responder positivamente a 2 questões, ela deve ser classificada como "Suspeita", entre 3 e 4 como "Cúmplice" e 5 como "Assassino". Caso contrário, ele será classificado como "Inocente".

Precedência de Operadores

primeiro



último

**	Exponencial
*, /, //, %	Multiplicação, divisão, resto
+, -	Adição e Subtração
in, is, is not, <, <=, >, >=, !=, ==	Comparações
not	NÃO
and	E
or	OU
if - elif - else	Expressões condicionais

Estruturas de Repetição

Repetições

- São utilizadas para executar **várias vezes** a mesma parte do programa
- Normalmente dependem de uma condição
- Repetições são a base de vários programas!

Estruturas de Repetição

Comando *while*

Comando **while**

- O comando **while** (enquanto) serve para executarmos alguma repetição **enquanto** uma condição for verdadeira (True)
- Sintaxe:

```
while <condição>:  
    #bloco que será repetido enquanto a condição for verdadeira
```

Comando **while**

```
x = 1  
while x <= 10:  
    print(x)  
    x = x + 1
```

1
2
3
4
5
6
7
8
9
10

Exemplo

- Fazer um programa para imprimir números sequenciais na tela, começando do número 1 até o número digitado pelo usuário.

```
Digite o último dígito da contagem: 5  
1  
2  
3  
4  
5
```

Exemplo

- Impressão do número 1 até o número digitado pelo usuário:

```
ultimo = int(input("Digite o último dígito da contagem: "))  
i = 1  
while i <= ultimo:  
    print(i)  
    i += 1
```

Equivalente a $i = i + 1$

Digite o último dígito da contagem: 5

1
2
3
4
5

- O lado direito do sinal de igual (=) é executado primeiro!
- O resultado é atribuído para a variável que estiver do lado esquerdo do sinal de igual (=)

while infinito

```
while True:  
    #bloco que sempre será executado,  
    #nunca sai do loop de repetição
```

Comando **break**

- Porém, mesmo quando utilizamos um **while** infinito, é possível que em determinadas situações o programa precise sair do loop de repetição.
- Esta interrupção pode ser alcançada com o comando **break**
- O comando **break** pode ser utilizado para interromper o while, independentemente da condição

Comando **break** - Exemplo

- Somatória de valores digitados pelo usuário até que o número 0 (zero) seja digitado; quando 0 for digitado o resultado da somatória é exibido:

```
somatoria = 0

while True:
    entrada = int(input("Digite um número a somar ou 0 para sair:"))
    if entrada == 0:
        break
    else:
        somatoria = somatoria + entrada

print("Somatória", somatoria)
```

```
Digite um número a somar ou 0 para sair:5
Digite um número a somar ou 0 para sair:6
Digite um número a somar ou 0 para sair:4
Digite um número a somar ou 0 para sair:0
Somatória 15
```

Estruturas de Repetição

Comando *for*

Comando **for**

- **for** é a estrutura de repetição mais utilizada
- Sintaxe:

```
for <referência> in <sequência>:  
    #bloco de código que será repetido  
    #a cada iteração
```

- Durante a execução, a cada iteração, a referência aponta para um elemento da sequência
- Uma vantagem do for com relação ao while é que o contador não precisa ser explícito!

Comando **for** - Exemplo

- Calcular a somatória dos números de 0 a 99

```
somatoria = 0  
  
for x in range(0,100):  
    somatoria = somatoria + x  
print(somatoria)
```

4950

A função **range(i, f, p)** é bastante utilizada nos laços com **for**

Ela gera um conjunto de valores inteiros:

- Começando de **i**
- Até valores menores que **f**
- Com passo **p**

Se o passo **p** não for definido, o padrão de 1 será utilizado.


Comando **continue** na repetição

- O comando **continue** funciona de maneira parecida com o **break**, porém o break interrompe e sai do *loop*;
- Já o **continue** faz com que a próxima iteração comece a ser executada, não importando se existem mais comandos depois dele ou não
- O **continue** não sai do *loop*
- O **continue** faz com que a próxima iteração seja executada imediatamente.

Comando **continue** na repetição

- Exemplo com **continue**

```
1 i = 0
2
3 while i < 12:
4     i+=2
5     if i == 8:
6         continue
7     print(i)
8 else:
9     print("Os números pares de 2 a 12 foram exibidos, com exceção do 8")
```



- Chama a próxima iteração
- Não executa os comandos da própria iteração: neste caso pula o `print()` com `i = 8`

```
2
4
6
10
12
Os números pares de 2 a 12 foram exibidos, com exceção do 8
```

Exercício 2

A sequência de números 0 1 1 2 3 5 8 13 21... é conhecida como Série de Fibonacci. Nesta sequência, cada número, depois dos 2 primeiros, é igual à soma dos 2 anteriores. Escreva um algoritmo que leia um inteiro N ($N < 46$) e mostre os N primeiros números dessa série.

Saída: Os valores devem ser mostrados na mesma linha, separados por um espaço em branco. Não deve haver espaço após o último valor.

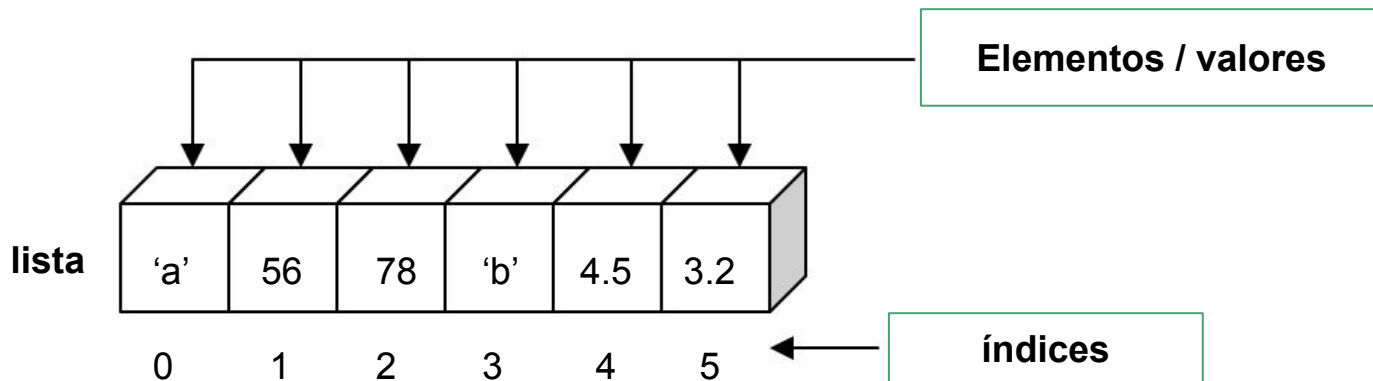
Exemplo:

Exemplo de Entrada	Exemplo de Saída
5	0 1 1 2 3

Listas

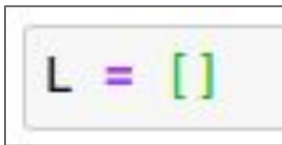
Lista

- Uma lista é uma variável que armazena um conjunto de valores
- Lista é um tipo de **variável** que permite o armazenamento de valores com tipos homogêneos ou heterogêneos (*do mesmo tipo ou de tipos diferentes*)
- Os valores armazenados em uma lista são acessados por um **índice**



Lista

- Para indicar que uma variável é uma lista, o símbolo de colchetes `[]` é utilizado para delimitar o conjunto
- Sintaxe - criando uma lista chamada `L`:



```
L = []
```

A screenshot of a code editor showing the Python syntax for creating an empty list. The variable name 'L' is in black, the assignment operator '=' is in purple, and the empty list notation '[]' is in green.

- `L` é uma lista vazia

Lista

- Exemplo:
 - Criando uma lista chamada **z** com 3 números inteiros

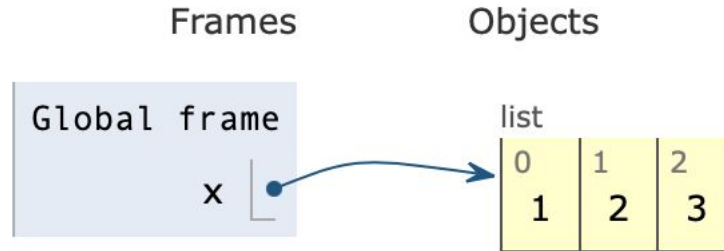
```
z = [5, 7, 1]
print(z)
```

[5, 7, 1]

- Dizemos que **z** tem tamanho **3**
- Podemos utilizar o *Python Tutor* para verificar melhor a lista:
 - <http://pythontutor.com/visualize.html#mode=edit>

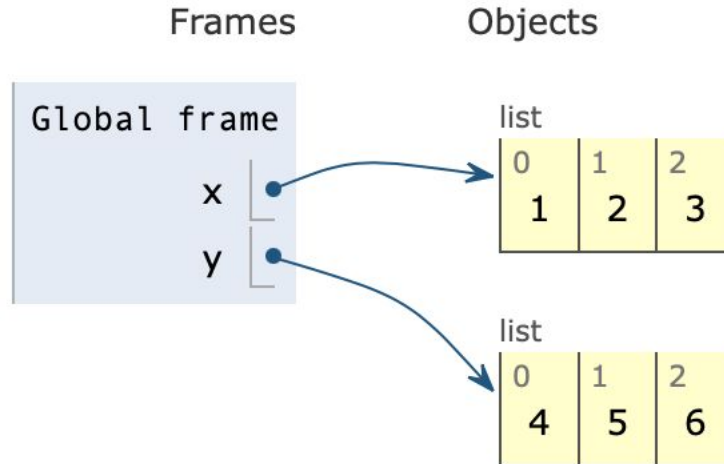
Python Tutor

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



Python Tutor

- `x = [1, 2, 3]`
- `y = [4, 5, 6]`



Lista - Exemplo

- Imprimindo a lista:

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

- Imprimindo a lista sem colchetes:

```
lista = [1, 2, 3, 4, 5]  
print(*lista)  
  
1 2 3 4 5
```

Lista - Tamanho da lista

- Como temos os métodos para incluir e remover dados das listas, nem sempre sabemos qual é o tamanho exato que a lista tem
- Para descobrirmos o tamanho da lista, utilizamos o método *len(lista)*

Exemplo:

```
a = [3, 4, 5]  
print(len(a))
```

```
a.append(9)  
a.append(11)  
print(len(a))
```

3

5

Lista - Acesso aos elementos

- Exemplo:

```
z = [5, 7, 1]
print(z)
```

[5, 7, 1]

- Para acessarmos o primeiro número da lista `z`, utilizamos a notação: `z[0]`
- Ou seja, da lista `z` queremos pegar o valor armazenado no índice `0`.

```
z = [5, 7, 1]
print(z[0])
print(z[1])
print(z[2])
```

5
7
1

Lista - Acesso aos elementos - lista `saldo`

<code>saldo</code>	
0	10.00
1	999.99
2	87.60
3	159.90
4	230.00

Exemplos:

```
saldo = [10.00, 999.99, 87.60, 159.90, 230.00]
```

- Mostrar terceiro item:

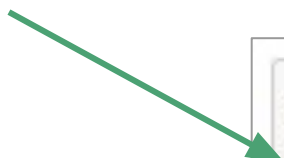
```
print( saldo[2] )
```

- Atribuir primeiro item a uma variável:

```
valor = saldo[0]
```

Lista - modificando elementos

- Utilizando o nome de uma lista com o índice desejado, podemos também modificar o conteúdo armazenado.
- Exemplo: Alterando o valor do primeiro elemento (*índice 0*) da lista **z**



```
z = [5, 7, 1]
z[0] = 32

print(z)

[32, 7, 1]
```

Lista - modificando elementos

estoque

0	123
1	64
2	100
3	26
4	555
5	975
6	87
7	3

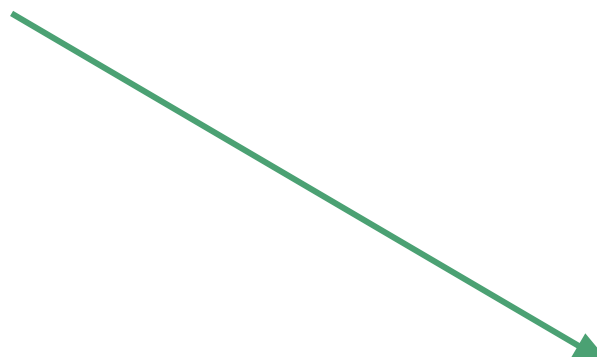
estoque = [123, 64, 100, 26, 555, 975, 87, 3]

- Substituir o último item:

estoque[7] = 21

estoque

0	123
1	64
2	100
3	26
4	555
5	975
6	87
7	21



Lista - Adicionando elementos no fim da lista

- Podemos ainda adicionar novos elementos no fim da lista
- Para isto, utilizamos o método *append(item)*
- Exemplo:

```
z = [32, 7, 1]  
print(z)
```

```
[32, 7, 1]
```

```
z.append("oi")  
print(z)
```

```
[32, 7, 1, 'oi']
```

Lista - Adicionando elemento em qualquer lugar

- Podemos ainda adicionar novos elementos em qualquer lugar da lista
- Para isto, utilizamos o método *insert(índice, item)*
- Exemplo:

```
z = [32, 7, 1]  
print(z)
```

```
[32, 7, 1]
```

```
z.insert(1, "oi")  
print(z)
```

```
[32, 'oi', 7, 1]
```

Lista - Removendo da lista pelo índice

- Podemos remover um elemento da lista
- Para isto, utilizamos o método *pop(índice)*
- Exemplo:

```
z = ["a", "b", "c", "d", "e"]  
print(z)
```

```
['a', 'b', 'c', 'd', 'e']
```

```
z.pop(1)  
print(z)
```

```
['a', 'c', 'd', 'e']
```

Lista - Removendo da lista pelo elemento

- Podemos remover um elemento da lista
- Para isto, utilizamos o método *remove(item)*
- Exemplos:

```
z = ["a", "b", "c", "d", "e"]  
print(z)
```

```
['a', 'b', 'c', 'd', 'e']
```

```
z.remove("d")  
print(z)
```

```
['a', 'b', 'c', 'e']
```

```
z = [1,2,3,1,4,5,1]  
z.remove(1)  
print(z)
```

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

Lista - Tamanho da lista

- Como temos os métodos para incluir e remover dados das listas, nem sempre sabemos qual é o tamanho exato que a lista tem
- Para descobrirmos o tamanho da lista, utilizamos o método *len(lista)*
- Exemplo:

```
a = [3, 4, 5]  
print(len(a))
```

```
a.append(9)  
a.append(11)  
print(len(a))
```

```
3  
5
```


inline for loop

Lista - Adicionando elementos usando for inline

- Podemos criar uma lista usando *for inline*:

```
lista = [ x for x in range(10)]  
print(lista)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Lista - Adicionando elementos usando for inline

- Podemos realizar variações usando range:

```
lista = [ x for x in range(0,11, 2)]  
print(lista)
```

```
[0, 2, 4, 6, 8, 10]
```

Lista - Adicionando elementos usando for inline

- Pode-se utilizar fórmulas para criar a lista:

```
lista = [ 2*x for x in range(6)]  
print(lista)
```

```
[0, 2, 4, 6, 8, 10]
```

Lista - Adicionando elementos usando for inline

- Pode-se adicionar estrutura de seleção inline:

```
lista = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
lista2 = [ x for x in lista if x % 2 == 0]  
print(lista2)
```

```
[0, 2, 4, 6, 8]
```

Exercício 3

Faça um código em Python que armazene 5 números inteiros informados pelo usuário em uma Lista. Exiba como saída o **MAIOR** número dessa Lista.

(Use laço de repetição para leitura da lista)

Exercício 4

Faça um programa para criar uma lista de 10 elementos (pedir para o usuário) e apresentar: a soma dos ELEMENTOS pares e a soma dos elementos de ÍNDICE par

(Use laço de repetição para leitura da lista)

Modularização: Funções

Funções

- Funções **são blocos de código** que realizam **determinadas tarefas** que normalmente precisam ser executadas diversas vezes dentro da mesma aplicação
- Assim, **tarefas muito utilizadas costumam ser agrupadas em funções**, que, depois de definidas, podem ser utilizadas / chamadas em qualquer parte do código somente pelo seu nome

Funções

- Já utilizamos algumas funções no Python:
 - `print("Olá!");`
 - `input("Digite o valor da entrada:")`
 - `int("3")`
 - `range(0,100)`
 - `randrange(0,10)`
 - `sleep(2)`
 - `sqrt(9)`

No Python: Funções - *def*

- Podemos criar / definir nossas próprias funções no Python utilizando a palavra-chave *def* seguido do *nome da função*, parênteses *()* e *:*
- Sintaxe:

```
def <nome da função>():  
    # tarefas que serão realizadas dentro da função
```

- Exemplo:

```
def imprimeOla():  
    print("Olá")
```

Funções **com** e **sem** parâmetros

- As funções podem ou não ter **parâmetros**, que **são valores enviados às funções** dentro dos parênteses no momento em que elas são chamadas
- Exemplos:

Sem parâmetros:

```
def soma():  
    a = 9  
    b = 8  
    print(a+b)
```

Chamada
da função

soma()

17

Com parâmetros:

```
def soma(a, b):  
    print(a+b)
```

soma(3,4)

Chamada
da função

7

Funções - *return*

- Além dos parâmetros, as funções podem ou não ter um **valor de retorno** (uma função sem retorno é chamada **procedimento**)
- O retorno é definido pela palavra-chave *return*
- Exemplos:

Sem parâmetros:

```
def soma():  
    a = 9  
    b = 8  
    return(a+b)  
  
print(soma())
```

17

Com parâmetros:

```
def soma(a, b):  
    return(a+b)  
  
print(soma(3,4))
```

7

Funções - *return* de múltiplos valores

- Colocar *vírgula* entre os valores de retorno
- A estrutura retornada será uma tupla
- Atribuir o retorno da função às variáveis desejadas
- Exemplo:

```
def minmax(a,b):  
    if a < b:  
        return a,b  
    else:  
        return b,a  
  
x = 10  
y = 2  
menor, maior = minmax(x,y)  
print("menor=%.2f , maior = %.2f" % (menor, maior))  
  
menor=2.00 , maior = 10.00
```

Funções - *return* de múltiplos valores

- Tuplas são similares às listas, porém são imutáveis!
- Tuplas não permitem adicionar, apagar, inserir ou modificar elementos
- **Tuplas são definidas com parêntesis (); Listas com colchetes []**
- Exemplo:

```
b = (2, 4, 5, 'tupla')  
print( b )
```

```
(2, 4, 5, 'tupla')
```

```
print( b[1] )
```

```
4
```

Exercício 5

Crie uma função que receba um valor inteiro em segundos e retorne uma tupla de três valores representando, respectivamente, horas, minutos e segundos

Exemplo: **converteSeg(121)**

Deve retornar: 0, 2, 1

Funções *lambda*

- No Python, podemos criar funções simples em somente uma linha
- Estas funções, ou expressões, são chamadas de *lambda*
- Exemplo: função *lambda* que recebe um parâmetro *x* e retorna o quadrado deste número.
 - *A função será associada a variável a*

```
a = lambda x: x**2  
print(a(3))
```

9

Funções *lambda*

- Exemplo:
- Função *lambda* que calcula o aumento, dado o valor inicial e a porcentagem de aumento:

```
aumento = lambda a,b : a*b/100  
aumento(100,5)
```

5.0

Exercício 6

Escreva uma função **lambda** com parâmetros que retorne o maior de dois números. A função deve se chamar **maximo(x, y)**.

Exercício 7

Faça uma função que receba quatro valores: I, A, B e C. Destes Valores, I é um valor inteiro valendo 1, 2 ou 3. A, B e C são valores reais. Escreva os números A, B e C obedecendo à tabela a seguir, dependendo do valor de I

Valor de I	Forma a Escrever
1	A, B e C em ordem crescente
2	A, B e C em ordem decrescente
3	O maior fica entre os outros dois números

Dicionários

Dictionaries

Dicionários

- O Dicionário é uma **estrutura de dados**
 - Estruturas de dados são maneiras de organizar os dados
 - **Listas** e **Tuplas** também são estruturas de dados
- Dicionários (em Python) são **vetores associativos**
- Vetores associativos são coleções **desordenadas** de dados, usadas para **armazenar** valores como um **mapa**:
 - Por meio de elementos formados pelo par **chave** e **valor**

Dicionários

- Assim, diferentemente das listas ou tuplas, que contém um único valor como elemento, o dicionário contém o par **chave:valor** (**key:value**)
 - **Chave (key)**: serve para deixar o dicionário otimizado
 - **Valor (value)**: valor do elemento associado a uma chave

Dicionários

- Dicionários diferem das listas essencialmente na maneira como os elementos são acessados:
 - **Listas**: valores são acessados por sua posição dentro da lista, via índice
 - **Dicionários**: valores são acessados por meio de suas **chaves** (*keys*)

Dicionários

- Um dicionário em Python funciona de forma semelhante ao dicionário de palavras:
 - As **chaves** (*keys*) de um dicionário devem ser exclusivas e com o tipo de dados imutáveis, como strings, inteiros ou tuplas
 - Porém, os **valores** (*values*) associados às chaves podem ser repetidos e de qualquer tipo

Dicionários

- Para criarmos um dicionário, devemos incluir uma sequência de elementos dentro de chaves `{ }`, separados por vírgula.
- A chave e o valor são separados por dois pontos `:`
- Cada elemento do dicionário é um par composto por chave (*key*) e valor (*value*).
- Sintaxe:

```
d = {  
    <key1>:<value1>,  
    <key2>:<value1>,  
    <key3>:<value2>  
}
```

Dicionários - Exemplos

Criando um dicionário com chaves inteiras

```
dicionario = {  
    1 : 'exemplo',  
    2 : 'de',  
    3 : 'dicionario'  
}
```

```
print(dicionario)
```

```
{1: 'exemplo', 2: 'de', 3: 'dicionario'}
```

Dicionários - Exemplos

Criando um dicionário com chaves de tipos mistos

```
teste = {  
    'nome' : 'fulano',  
    5 : 'cinco',  
    'lista' : [1, 2, 4]  
}
```

```
print(teste)
```

```
{'nome': 'fulano', 5: 'cinco', 'lista': [1, 2, 4]}
```

Dicionários - Acessando elementos

- Os valores são acessados por meio de suas chaves
- Utiliza-se o nome do dicionário e a chave dentro de colchetes []

```
ingles = {  
    'um' : 'one',  
    'dois' : 'two',  
    'tres' : 'three',  
    'quatro' : 'four',  
    'cinco' : 'five'  
}
```

```
ingles['um']
```

'one'

```
ingles['quatro']
```

'four'

```
ingles_num = {  
    1 : 'one',  
    2 : 'two',  
    3 : 'three',  
    4 : 'four',  
    5 : 'five'  
}
```

```
ingles_num[3]
```

'three'

```
ingles_num[2]
```

'two'

Dicionários - Adicionando novos elementos

- Para adicionar um novo elemento a um dicionário existente, basta atribuir o novo valor e especificar a chave dentro de colchetes

Dicionário original

```
ingles_num = {  
    1 : 'one',  
    2 : 'two',  
    3 : 'three',  
    4 : 'four',  
    5 : 'five'  
}
```

```
ingles_num[7] = 'seven'
```

Acrescentando um novo elemento: chave = 7 e valor = 'seven'

```
print(ingles_num)
```

```
{1: 'one', 2: 'two', 3: 'three', 4: 'four', 5: 'five', 7: 'seven'}
```

Dicionários - Removendo elementos

- Para deletar um elemento de um dicionário utilizamos a palavra-chave **del**

Dicionário original

```
ingles_num = {  
    1 : 'one',  
    2 : 'two',  
    3 : 'three',  
    4 : 'four',  
    5 : 'five'  
}
```

```
del ingles_num[3]
```

Removendo o elemento pela chave = 3

```
print(ingles_num)
```

```
{1: 'one', 2: 'two', 4: 'four', 5: 'five'}
```

Dicionários - Removendo elementos

- Para deletar um elemento de um dicionário utilizamos a palavra-chave **del**

Dicionário original

```
ingles = {  
    'um' : 'one',  
    'dois' : 'two',  
    'tres' : 'three',  
    'quatro' : 'four',  
    'cinco' : 'five'  
}
```

```
del ingles['quatro']
```

Removendo o elemento pela chave = 'quatro'

```
print(ingles)
```

```
{'um': 'one', 'dois': 'two', 'tres': 'three', 'cinco': 'five'}
```


Dicionários - Alguns métodos

- **items()**: retorna todos os elementos do dicionário - pares chave:valor

```
dicionario = {  
    'um' : 'exemplo',  
    'dois' : 'de',  
    'tres' : 'dicionario'  
}
```

```
d = dicionario.items()
```

```
d = list(d)  
print(d)
```

```
[('um', 'exemplo'), ('dois', 'de'), ('tres', 'dicionario')]
```

Dicionários - Alguns métodos

- **keys()**: retorna todas as chaves do dicionário

```
diccionario = {  
    'um' : 'exemplo',  
    'dois' : 'de',  
    'tres' : 'dicionario'  
}
```

```
d = diccionario.keys()
```

```
d = list(d)  
print(d)
```

```
['um', 'dois', 'tres']
```

```
print(d[0])
```

```
um
```

Dicionários - Alguns métodos

- **values()**: retorna todas valores do dicionário

```
dicionario = {  
    'um' : 'exemplo',  
    'dois' : 'de',  
    'tres' : 'dicionario'  
}
```

```
d = dicionario.values()
```

```
d = list(d)  
print(d)
```

```
['exemplo', 'de', 'dicionario']
```

```
print(d[0])
```

```
exemplo
```

Dicionários - Iteração

- Podemos utilizar estruturas de repetição para iterar por um dicionário

```
dicionario = {  
    'um' : 'exemplo',  
    'dois' : 'de',  
    'tres' : 'dicionario'  
}
```

Iterando pelas chaves

```
for chave in dicionario:  
    print(chave)
```

um
dois
tres

Iterando pelos valores

```
for valor in dicionario.values():  
    print(valor)
```

exemplo
de
dicionario

Dicionários - Iteração

- Podemos utilizar estruturas de repetição para iterar por um dicionário

```
dicionario = {  
    'um' : 'exemplo',  
    'dois' : 'de',  
    'tres' : 'dicionario'  
}
```

```
for chave, valor in dicionario.items():  
    print(chave, valor)
```

```
um exemplo  
dois de  
tres dicionario
```

Exercício 8

Escreva uma função chamada *procuraReversa* que encontre todas as chaves, em um dicionário, que estão associadas a um valor específico. A função receberá o dicionário e o valor a procurar como seus únicos parâmetros. A função retornará uma lista (possivelmente vazia) de chaves associadas ao valor fornecido. Faça um programa principal que mostra o funcionamento da função. Seu programa principal deve criar um dicionário e mostrar que a função *procuraReversa* funciona corretamente quando retorna várias chaves, uma única chave ou nenhuma chave.

Módulos

Modules

Módulos

- Para o Python, **Módulos** são arquivos fonte que podem ser importados para um programa
- Podem conter qualquer estrutura do Python e são executados quando importados
- Um módulo é um arquivo **.py** que pode conter:
 - Funções
 - Variáveis e Constantes
 - Classes

Módulos

- Os módulos são carregados através da instrução *import*.
- Desta forma, ao usar alguma estrutura do módulo, é necessário identificá-lo.
 - Isto é chamado de *importação absoluta*

Módulos

- Exemplo de *importação absoluta*:

```
1 import os
2
3 os.remove('arquivo.txt')
```

Módulos

- Também é possível importar de forma *relativa*, utilizando as palavras-chaves *from* e *import*
- Exemplo:

```
1 from os import remove
2
3 remove('arquivo.txt')
```

Módulos

- Ainda usando a *importação relativa*, podemos usar o *** para importar tudo o que está no módulo.
- Exemplo:

```
1 from os import *  
2  
3 remove('arquivo.txt')
```

Módulos

- A ***importação relativa*** com *** não** é recomendada, pois pode gerar problemas, como a ofuscação de variáveis.

Programação Orientada a Objetos

Alguns paradigmas de programação

- Programação Imperativa / Estruturada (C, FORTRAN)
- Programação Orientada a Objetos (C++, Java, Python)
- Programação Funcional (LISP, Haskell)
- Programação Declarativa (SQL)
- Programação Lógica (Prolog)

OBS: Várias linguagens são multiparadigmas, ou seja, podem ser programadas com diferentes paradigmas

POO

- **Programação Orientada a Objetos (POO)** é um **paradigma** fundamental de programação
- A ideia é abstrair um programa como uma coleção de objetos que interagem entre si
- Duas das grandes razões para o uso do paradigma são:
 - A possibilidade de **reutilização de código** e
 - A melhoria na **organização** de um projeto

POO

- O conceito formal de Orientação a Objetos foi introduzido em meados de 1960, com a linguagem **Simula 67** (Centro Norueguês de Computação em Oslo)
- O desenvolvimento completo da POO veio com o **Smalltalk 80** (1980)

Classes

- Representam itens do mundo real:
 - Exemplos:
 - Pessoas
 - Veículos
 - Robôs
- São Compostas de:
 - **Atributos** (variáveis de instância)
 - **Métodos** (funções-membro)

Objetos

- Todo objeto pertence a uma **classe**
- Representam **instâncias** de entidades no mundo real
- São criados a partir das classes
- São instâncias das classes
- Os objetos associam valores específicos aos atributos

Instanciar (Informática)

- Instanciar é criar um objeto, ou seja, alocar um espaço na memória, para posteriormente poder utilizar os métodos e atributos que o objeto dispõe
- Em informática instância é usada com o sentido de exemplar. No contexto da orientação ao objeto, instância significa a concretização de uma classe.

Classes e Objetos

Quando definimos uma **classe** de objetos, estamos, na verdade, definindo que **propriedades** e **métodos** o **objeto possui!**

Diferença entre Classe e Objeto

- **Classe:**

- É um modelo
- De maneira mais prática, é como se fosse a **planta de uma casa**

- **Objeto:**

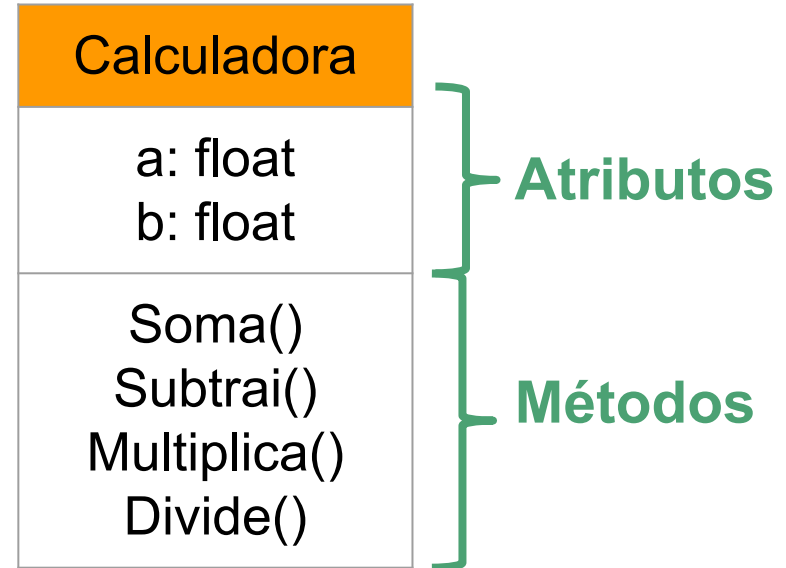
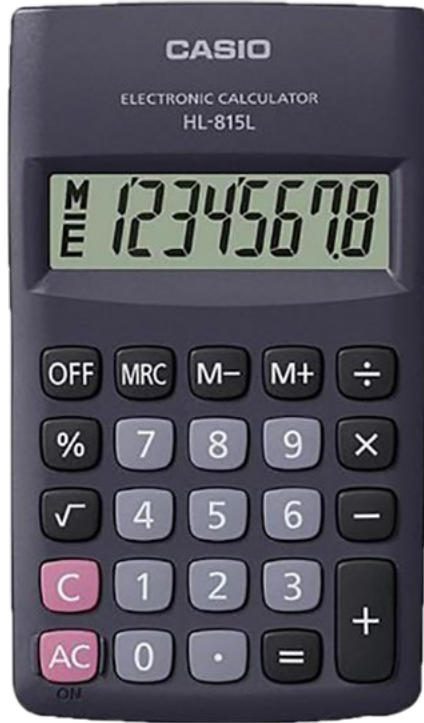
- É criado a partir da classe
- É como se fosse a **própria casa** construída
- Pode-se construir várias casas a partir da mesma planta, assim como podemos instanciar vários objetos de uma só classe

Estrutura de uma Classe

Nome da Classe
- Atributos
- Métodos

- **Atributos** são **variáveis** que armazenam informações do objeto.
- **Métodos** são as operações (**funções**) que o objeto pode realizar.

Exemplo de Classe



Exemplo de Classe e Objetos

Cat
size: float color: string positionX: float positionY: float
moveForward() moveBackward() moveUP() moveDown()

```
Cat garfield;  
Cat tom;  
Cat felix ;  
Cat scratchy;
```



Algumas Linguagens Orientadas a Objetos



POO no Python

```
1 | class NomeClasse:  
2 |     # atributos  
3 |  
4 |     # métodos
```

```
1 | class Aluno:  
2 |     nome = ""  
3 |     ra = 0  
4 |  
5 |     def mostraAluno(self):  
6 |         print("Nome: %s" % self.nome)  
7 |         print("R.A.: %d" % self.ra)
```

POO no Python

self serve para identificar os atributos e os métodos da classe

Serve principalmente para delimitar o escopo e tirar ambiguidade com outras possíveis variáveis

```
1 class Aluno:
2     nome = " "
3     ra = 0
4
5     def mostraAluno(self):
6         print("Nome: %s" % self.nome)
7         print("R.A.: %d" % self.ra)
8
9 aluno = Aluno()
10 aluno.nome = "Danilo"
11 aluno.ra = 123456789
12 aluno.mostraAluno()
13
14
```

```
Nome: Danilo
R.A.: 123456789
```

POO no Python

cria o objeto aluno
da classe **Aluno**

```
1 class Aluno:
2     nome = " "
3     ra = 0
4
5     def mostraAluno(self):
6         print("Nome: %s" % self.nome)
7         print("R.A.: %d" % self.ra)
8
9     aluno = Aluno()
10    aluno.nome = "Danilo"
11    aluno.ra = 123456789
12    aluno.mostraAluno()
13
14
```

```
Nome: Danilo
R.A.: 123456789
```

POO no Python

executa o método mostraAluno() do objeto aluno

```
1 class Aluno:
2     nome = " "
3     ra = 0
4
5     def mostraAluno(self):
6         print("Nome: %s" % self.nome)
7         print("R.A.: %d" % self.ra)
8
9 aluno = Aluno()
10 aluno.nome = "Danilo"
11 aluno.ra = 123456789
12 aluno.mostraAluno()
13
14
```

```
Nome: Danilo
R.A.: 123456789
```

POO no Python - Construtor

No Python, o construtor da classe é chamado de `__init__`

```
1 class Aluno:
2
3     def __init__(self, nome, ra):
4         self.nome = nome
5         self.ra = ra
6
7     def mostraAluno(self):
8         print("Nome: %s" % self.nome)
9         print("R.A.: %d" % self.ra)
10
11 aluno = Aluno("Danilo", 123456789)
12 aluno.mostraAluno()
13
14
15
16
```

construtor:
inicializa o valor
dos **atributos**
quando o objeto é
instanciado

Cria / instancia o objeto
aluno já com valores
específicos para os
atributos **nome** e **ra**

```
Nome: Danilo
R.A.: 123456789
```

Entrega

Exercício para Entrega

- As entregas devem ser feitas no formato ***ipynb*** (IPython Notebook - Jupyter Notebook)
- Os códigos devem ser documentados e explicados (utilizando células de Markdown)
- Plágios serão punidos com máximo rigor!

Exercício para Entrega

Faça um programa, utilizando o paradigma de Orientação a Objetos, para simular 1.000 lançamentos de dois dados (6 lados cada um). Então, obtenha a somatória obtida pelos dois dados. Você deve contar o número de vezes que cada somatória acontece. O seu programa deve, então, exibir uma tabela que resume esses resultados. Mostre a frequência para cada resultado como uma porcentagem do número total de lançamentos. Prove matematicamente (com o uso de probabilidade) que o código desenvolvido resolve adequadamente o problema.

Exercício para Entrega

Uma proposta de desenvolvimento (*não precisa ser assim*):

- Crie a classe **Dado**
- A classe pode ter os atributos **numeroDeLados** e **valor**
- A classe pode ter o método **lançar()**, que obtém um novo valor aleatório para o dado
- Você deve instanciar dois dados, lançar ambos, obter a somatória e armazenar o valor para conseguir emitir o relatório final solicitado.