



Aula 2
Introdução ao
Python - Parte II

Funções

- **Funções:** utilizadas para definir uma sequência de códigos que pode ser utilizada repetidas vezes
- Notação:

```
def [function_name] ([arguments]):  
    [function body]  
    return [value] #Opcional
```

- Definindo uma função para calcular o IMC:

```
In [1]: def calcula_imc(peso, altura):  
        valor = peso / (altura ** 2)  
        return valor
```

```
In [2]: calcula_imc(94,1.93)
```

```
Out[2]: 25.235576794007894
```

Funções

- Como tudo é objeto em Python, e objetos são uma referência à uma região de memória, a passagem de argumentos para funções é automaticamente por referência

```
In [6]: def add_cidades(lista, cidade_nova):  
        lista.append(cidade_nova)
```

```
In [7]: lista = ['Porto Ferreira']
```

```
In [8]: add_cidades(lista, 'Três Lagoas')
```

```
In [9]: lista
```

```
Out[9]: ['Porto Ferreira', 'Três Lagoas']
```

Funções

- Entretanto, o Python possui tipos de objetos imutáveis: `int`, `float`, `string`, `tuple`
- Isso significa que não há alteração de objetos desses tipos no corpo das funções (equivalente à passagem por valor)

```
In [10]: def soma_10(valor):  
         valor += 10
```

```
In [11]: valor = 10
```

```
In [12]: soma_10(valor)
```

```
In [13]: valor
```

```
Out[13]: 10
```

Funções

- Pode-se retornar mais de um valor com uma função
- Para isso, basta retornar os múltiplos valores separados por vírgula (e receber esses múltiplos valores em múltiplas variáveis, também separadas por vírgula)

```
In [15]: def calcula_area_perimetro_ret(base, altura):  
         area = base * altura  
         perimetro = (base * 2) + (altura * 2)  
         return area, perimetro
```

```
In [16]: area, perimetro = calcula_area_perimetro_ret(5,10)
```

```
In [17]: area
```

```
Out[17]: 50
```

```
In [18]: perimetro
```

```
Out[18]: 30
```

Funções

- Pode-se definir valores padrões para as funções, de forma que não seja necessário informá-los durante as chamadas da função
- Para isso, basta atribuir um valor ao argumento na declaração da função

```
In [36]: 1 def area_cubo(base=1, altura=2, largura=3):  
        2     return base*altura*largura
```

```
In [38]: 1 area_cubo() #invocando a função sem passar argumentos
```

```
Out[38]: 6
```

```
In [39]: 1 area_cubo(largura=5) #invocando a função só informando a largura
```

```
Out[39]: 10
```

Funções

- As funções podem ter argumentos variáveis
- Para isso, basta utilizar o símbolo `*` precedendo um argumento que representará uma “lista” variável de argumentos

```
In [41]: 1 def soma(*numeros):  
2         resultado = 0  
3         for num in numeros:  
4             resultado += num  
5         return resultado
```

```
In [42]: 1 soma(1,2)
```

```
Out[42]: 3
```

```
In [43]: 1 soma(1,2,3)
```

```
Out[43]: 6
```

```
In [44]: 1 soma(1,2,3,4)
```

```
Out[44]: 10
```

Funções

- Também é possível criar funções utilizando o conceito de *lambda function*
- Por meio das *lambda functions*, é possível criar funções em uma linha
- A sintaxe é `lambda arguments : expression`

```
In [1]: 1 soma = lambda num1, num2 : num1 + num2  
        2  
        3 print(soma(4,5))  
9
```

- O valor da expressão será automaticamente retornado pela função

Funções

- Pode-se “desempacotar” valores de uma variável heterogênea para passar argumentos para funções utilizando o símbolo `*` ou `**` ao lado da variável dependendo da estrutura da variável

```
In [2]: 1 def function(par1, par2, par3):  
2         print('par1:', par1)  
3         print('par2:', par2)  
4         print('par3:', par3)
```

```
In [8]: 1 params = {'par1': 'abacate',  
2                 'par3': 'pera',  
3                 'par2': 'maçã'}  
4
```

```
In [9]: 1 function(**params)  
  
par1: abacate  
par2: maçã  
par3: pera
```

```
In [11]: 1 params2 = ['pera', 'maçã', 'abacate']
```

```
In [14]: 1 function(*params2)  
  
par1: pera  
par2: maçã  
par3: abacate
```

Criando Módulos

- Módulos são úteis para agrupar funções ou classes relacionadas
- Cada módulo corresponde à um arquivo e o nome do arquivo será o nome do módulo
- Portanto, para criar um módulo basta criar um arquivo `.py` com as várias variáveis e funções definidas dentro desse arquivo

Criando Módulos

```
geometria.py* X
1#!/usr/bin/env python3
2# -*- coding: utf-8 -*-
3"""
4Created on Sun Jul 28 10:46:00 2019
5
6@author: rafael
7"""
8
9pi = 3.14
10
11def calcula_area_quadrilatero(base, altura):
12    return base * altura
13
14def calcula_perimetro_quadrilatero(base, altura):
15    return (base * 2) + (altura * 2)
16
17def calcula_area_circulo(raio):
18    return pi * raio ** 2
19
20def calcula_perimetro_circulo(raio):
21    return 2 * pi * raio
```

Importando Módulos

- Para importar um módulo basta usar a palavras-chave `import` e o nome do módulo
- Para utilizar as variáveis ou funções do módulo:
`[nome_modulo].[variavel_ou_função]`

```
In [21]: 1 import geometria
```

```
In [22]: 1 geometria.calcula_area_circulo(2)
```

```
Out[22]: 12.56
```

```
In [23]: 1 geometria.calcula_area_quadrilatero(3,4)
```

```
Out[23]: 12
```

Importando Módulos

- O Python provê a possibilidade de importar uma única função ou variável do módulo
- Ao fazer isso, é possível referenciar-se diretamente ao elemento, sem a necessidade do nome do módulo precedê-lo

```
In [27]: 1 from geometria import calcula_area_quadrilatero
```

```
In [28]: 1 calcula_area_quadrilatero(3,4)
```

```
Out[28]: 12
```

Importando Módulos

- O Python provê uma funcionalidade de “renomear” ou “dar um apelido” (*alias*) à um módulo de forma a facilitar o seu uso
- Para isso, basta utilizar a palavra-chave *as* e em seguida o *alias* do módulo

```
In [24]: 1 import geometria as geo
```

```
In [25]: 1 geo.calcula_area_circulo(2)
```

```
Out[25]: 12.56
```

```
In [26]: 1 geo.calcula_area_quadrilatero(3,4)
```

```
Out[26]: 12
```

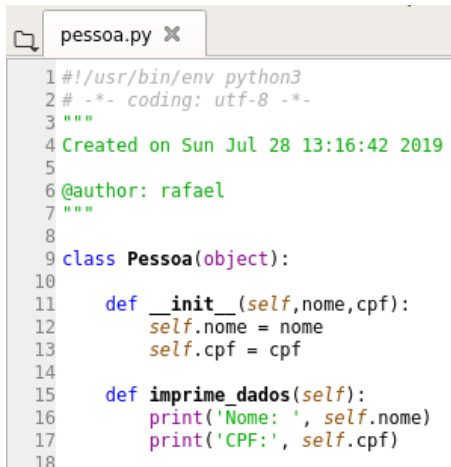
Orientação à Objetos

- Python provê recurso de orientação à objetos, como definição de classes, métodos e herança, porém não possui todos os componentes de orientação a objetos vistos em outras linguagem quanto à um amplo recurso de encapsulamentos
- Objetos não normalmente definidos dentro de módulos
- A uso de objetos em módulo é semelhante ao uso de funções em módulo

Definindo uma Classe

- Notação: `class [nome_classe](object):`
- O `Object` como argumento da classe é usado para explicitar que toda classe realiza uma herança com a classe `object` (não é obrigatório)
- A notação do método construtor é dada por: `def __init__(argumentos):`
- O primeiro argumento de qualquer método é `self` que é utilizado para se referenciar ao próprio objeto
- Todos os atributos da classe são definidos dentro do método construtor

Definindo uma Classe



```
pessoa.py X
1#!/usr/bin/env python3
2# -*- coding: utf-8 -*-
3"""
4Created on Sun Jul 28 13:16:42 2019
5
6@author: rafael
7"""
8
9class Pessoa(object):
10
11    def __init__(self, nome, cpf):
12        self.nome = nome
13        self.cpf = cpf
14
15    def imprime_dados(self):
16        print('Nome: ', self.nome)
17        print('CPF:', self.cpf)
18
```

Criando um Objeto

- Para criar um objeto, basta utilizar a notação:
`[nome_classe]([argumentos_construtor])`
- Para acessar os campos e métodos, basta utilizar a notação:
`[identificador_objeto].[campo_ou_classe]`

```
In [29]: 1 from pessoa import Pessoa
```

```
In [30]: 1 p1 = Pessoa('Rafael', '000.000.000-00')
```

```
In [31]: 1 p1.imprime_dados()
```

```
Nome: Rafael  
CPF: 000.000.000-00
```

OBSERVAÇÃO: não é necessário informar ou passar o argumento correspondente ao `self`

Herança

- Para realizar uma herança, basta utilizar o nome da classe a ser herdada como argumento da classe a ser criada
- Para se referenciar à superclasse, basta utilizar a notação: `super([nome_classe_filha], self).[campo_ou_método]`

```
9 class Pessoa(object):
10
11     def __init__(self, nome, cpf):
12         self.nome = nome
13         self.cpf = cpf
14
15     def imprime_dados(self):
16         print('Nome: ', self.nome)
17         print('CPF:', self.cpf)
18
19
20 class Aluno(Pessoa):
21
22     def __init__(self, nome, cpf, rga, curso):
23         super(Aluno, self).__init__(nome, cpf)
24         self.curso = curso
25         self.rga = rga
26
27     def imprime_dados(self):
28         super(Aluno, self).imprime_dados()
29         print('RGA:', self.rga)
30         print('Curso:', self.curso)
```

Herança

```
In [1]: 1 from pessoa import Aluno
```

```
In [2]: 1 aluno = Aluno('Rafael', '000.000.000-00', '20192545', 'SI')
```

```
In [3]: 1 aluno.imprime_dados()
```

```
Nome: Rafael  
CPF: 000.000.000-00  
RGA: 20192545  
Curso: SI
```

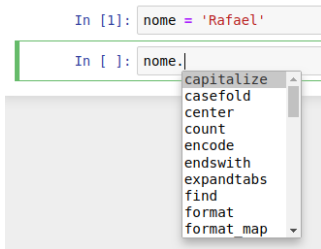
OBSERVAÇÃO: Python suporta herança múltipla. Para isso basta informar as múltiplas classes como argumento da classe separadas por vírgula. Caso haja conflito de métodos, será dada prioridade aos métodos das classes mais à esquerda.

Métodos Especiais

- Existem alguns métodos que não são “enxergados” a princípio mas que existem dentro dos objetos e que o Python faz uso deles em algumas situações
- O nome desses métodos é caracterizado por dois underlines, seguido dos nomes dos métodos, e mais dois underlines

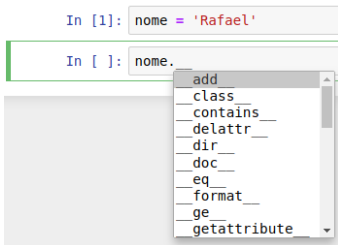
```
In [1]: nome = 'Rafael'
```

```
In [ ]: nome.
```



```
In [1]: nome = 'Rafael'
```

```
In [ ]: nome.
```



Métodos Especiais

- Por exemplo, quando é utilizado o operador de comparação “==”, é invocado o método `__eq__(self, outro)`, na qual o objeto à esquerda do operador corresponderá ao `self` e o objeto à direita corresponderá ao `outro`

```
In [1]: nome = 'Rafael'

In [2]: from pessoa import Pessoa

In [3]: pessoa1 = Pessoa('Rafael', 'XXXXX')

In [4]: pessoa2 = Pessoa('Rafael', 'XXXXX')

In [5]: """Como o Python não foi implementado o método __eq__,
         o Python não sabe como comparar os objetos. Com isso,
         será retornado Falso mesmo se os dois objetos
         contiverem os mesmo valores dos campos"""
         pessoa1 == pessoa2

Out[5]: False
```

Métodos Especiais

- Porém, ao implementar o método `__eq__`, o resultado obtido é diferente

```
9 class Pessoa(object):
10
11     def __init__(self,nome,cpf):
12         self.nome = nome
13         self.cpf = cpf
14
15     def imprime_dados(self):
16         print('Nome: ', self.nome)
17         print('CPF:', self.cpf)
18
19     def __eq__(self,outro):
20         if self.cpf == outro.cpf and self.nome == outro.nome:
21             return True
22         else:
23             return False
24
```

Métodos Especiais

```
In [1]: nome = 'Rafael'
```

```
In [2]: from pessoa import Pessoa
```

```
In [3]: pessoa1 = Pessoa('Rafael', 'XXXXX')
```

```
In [4]: pessoa2 = Pessoa('Rafael', 'XXXXX')
```

```
In [5]: """agora como o método __eq__(...) foi implementado,  
        o resultado será diferente"""  
        pessoa1 == pessoa2
```

```
Out[5]: True
```


Métodos Especiais

- Ao invocar o método `print(...)`, é chamado automaticamente o método `__str__(...)` e ao invocar o método `len(...)`, é invocado automaticamente o método `__len__(...)`

```
9 class Pessoa(object):
10
11     def __init__(self, nome, cpf):
12         self.nome = nome
13         self.cpf = cpf
14
15     def imprime_dados(self):
16         print('Nome: ', self.nome)
17         print('CPF:', self.cpf)
18
19     def __eq__(self, outro):
20         if self.cpf == outro.cpf and self.nome == outro.nome:
21             return True
22         else:
23             return False
24
25     def __str__(self):
26         resultado = "Nome: " + self.nome
27         resultado += "\nCPF: " + self.cpf
28         return resultado
29
30     def __len__(self):
31         tamanho = len(self.nome) + len(self.cpf)
32         return tamanho
```

Métodos Especiais

```
In [2]: from pessoa import Pessoa
```

```
In [3]: pessoa1 = Pessoa('Rafael', 'XXXXX')
```

```
In [4]: pessoa2 = Pessoa('Rafael', 'XXXXX')
```

```
In [5]: """Como o Python não foi implementado o método __eq__,  
o Python não sabe como comparar os objetos. Com isso,  
será retornado Falso mesmo se os dois objetos  
contiverem os mesmo valores dos campos"""  
pessoa1 == pessoa2
```

```
Out[5]: True
```

```
In [6]: len(pessoa1)
```

```
Out[6]: 11
```

```
In [7]: print(pessoa1)
```

```
Nome: Rafael  
CPF: XXXXX
```

Encapsulamento

- O encapsulamento dos membros da classe se dá por meio da utilização do “_” ou “__” antes dos nomes dos campos e métodos
- Depois disso, basta prover métodos get e set, ou ainda utilizar as notações `property` ou `[nome_da_variavel].setter` para simular o acesso e atribuição de valor à um campo, mas que na verdade irá invocar um método definido

Encapsulamento

```
In [4]: 1 class Pessoa(object):  
2  
3     def __init__(self, nome, cpf):  
4         self.__nome = nome  
5         self.__cpf = cpf  
6  
7     def get_nome(self):  
8         return self.__nome  
9  
10    def get_cpf(self):  
11        return self.__cpf  
12  
13    def set_nome(self, nome):  
14        self.__nome = nome  
15  
16    def set_cpf(self, cpf):  
17        self.__cpf = cpf
```

```
In [5]: 1 p1 = Pessoa('Rafael', '00000000')
```

```
In [6]: 1 p1.get_nome()
```

```
Out[6]: 'Rafael'
```

```
In [7]: 1 p1.set_nome('Ronaldo')
```

```
In [8]: 1 p1.get_nome()
```

```
Out[8]: 'Ronaldo'
```

Encapsulamento

```
In [15]: 1 class Pessoa(object):
2
3     def __init__(self, nome, cpf):
4         self.__nome = nome
5         self.__cpf = cpf
6
7     @property
8     def nome(self):
9         return self.__nome
10
11     @property
12     def cpf(self):
13         return self.__cpf
14
15     @nome.setter
16     def nome(self, nome):
17         self.__nome = nome
18
19     @cpf.setter
20     def cpf(self, cpf):
21         self.__cpf = cpf
```

```
In [16]: 1 p1 = Pessoa('Rafael', '000000000')
```

```
In [17]: 1 p1.nome
```

```
Out[17]: 'Rafael'
```

```
In [18]: 1 p1.nome = 'Ronaldo'
```

```
In [19]: 1 p1.nome
```

```
Out[19]: 'Ronaldo'
```

Campos e Métodos Estáticos

- Para definir um campo estático, basta declará-lo fora do método construtor
- Para definir um método estático, utilize a notação `staticmethod` antes da declaração do método
- Para acessar os campos e métodos estáticos, basta utilizar o nome da classe, isto é, `[Nome da classe].[campo ou método]`

Campos e Métodos Estáticos

```
In [23]: 1 class Pessoa(object):
2
3     descricao = 'Essa é uma classe para armazenar informações pessoais'
4
5     def __init__(self, nome, cpf):
6         self.__nome = nome
7         self.__cpf = cpf
8
9     def imprime_descricao():
10         print(f'Descrição da class: {Pessoa.descricao}')
11
```

```
In [24]: 1 Pessoa.descricao
```

```
Out[24]: 'Essa é uma classe para armazenar informações pessoais'
```

```
In [25]: 1 Pessoa.imprime_descricao()
```

```
Descrição da class: Essa é uma classe para armazenar informações pessoais
```

Polimorfismo

- Uma vez que o Python é uma linguagem dinamicamente tipada, o simples uso do conceito *Duck Typing* permite o comportamento polimórfico
 - Programar no geral
 - Ter diferentes comportamentos em um processamento genérico
- O conceito *Duck Typing* diz que se algo se parece como pato e se comporta como pato, então provavelmente é um pato
- Com isso, dado que diferentes classes implementem o método com a mesma assinatura (nome do método e quantidade de parâmetros), é possível fazer o processamento polimórfico considerando essas classes

Campos e Métodos Estáticos

```
1 class Pessoa(object):
2
3     def __init__(self, nome, cpf):
4         self.__nome = nome
5         self.__cpf = cpf
6
7     def imprime_dados(self):
8         print(f'Nome: {self.__nome}')
9         print(f'CPF: {self.__cpf}')
```

```
1 class Impressora(object):
2
3     def __init__(self, marca, modelo):
4         self.__marca = marca
5         self.__modelo = modelo
6
7     def imprime_dados(self):
8         print(f'Marca: {self.__marca}')
9         print(f'Modelo: {self.__modelo}')
```

```
1 lista = [Pessoa('Rafael', '00000'),
2          Pessoa('Ricardo', '111111'),
3          Impressora('HP', 'HP Deskjet 2776')]
```

```
1 for obj in lista:
2     print('=====' )
3     obj.imprime_dados()
```

```
=====
Nome: Rafael
CPF: 00000
=====
Nome: Ricardo
CPF: 111111
=====
Marca: HP
Modelo: HP Deskjet 2776
```

Manipulando Strings

- A classe *string* possui uma série de facilidades para manipulação e padronização de *strings*

```
In [26]: str_teste = 'Tópicos em Inteligência'

In [27]: str_teste = str_teste + ' Artificial' #Concatenando strings

In [28]: str_teste
Out[28]: 'Tópicos em Inteligência Artificial'

In [29]: str_teste.upper() #Retornando a string apenas com letras maiúsculas
Out[29]: 'TÓPICOS EM INTELIGÊNCIA ARTIFICIAL'

In [30]: str_teste.lower() #Retornando a string apenas com letras minúsculas
Out[30]: 'tópicos em inteligência artificial'

In [31]: str_teste.count('ic') #Conta quantas vezes a substring 'ic' ocorre
Out[31]: 2

In [35]: #Substituindo uma string por outra
str_teste.replace('Artificial', 'Computacional')
Out[35]: 'Tópicos em Inteligência Computacional'

In [36]: #retorna uma lista de strings separadas pelo caractere informado
str_teste.split(' ')
Out[36]: ['Tópicos', 'em', 'Inteligência', 'Artificial']
```

Manipulando String

- Para fazer uso de expressões regulares, deve-se importar o pacote `re`
- Um dos métodos mais utilizados é o `findall(arg1, arg2)`, em que o primeiro argumento é a expressão regular e o segundo é a *string* a ser analisada
- O retorno do método é uma lista com todos os padrões casados

```
In [37]: import re
In [38]: str_teste
Out[38]: 'Tópicos em Inteligência Artificial'

In [39]: #Gera uma lista com todos os resultados do casamento
re.findall(' [a-z]{2}', str_teste)
Out[39]: [' em ']

In [40]: re.findall('[A-Z]\\w+', str_teste)
Out[40]: ['Tópicos', 'Inteligência', 'Artificial']
```

Lendo e Escrevendo em Arquivos

- Sintaxes básicas:
 - Abrir: `identificador = open(caminho,modo)`
 - Fechar: `identificador.close()`
 - Modos:
 - `r`: abre para leitura
 - `w`: abre para escrita
 - `x`: abre para escrita, mas falha se o arquivo já existir
 - `a`: escrita do tipo *append*
 - `+`: abre tanto para escrita quanto para leitura
 - Por padrão a leitura e escrita é feita em modo texto
 - Ao acrescentar o caractere “b” no modos apresentados acima, a leitura e escrita é feita em modo binário

Leitura em Modo Texto

- Uma maneira simples de se ler um arquivo é utilizando o método `readlines()`, que irá retornar uma linha na qual cada posição corresponde à uma linha do arquivo

```
In [5]: 1 arquivo = open('batatinha.txt', 'r')

In [9]: 1 linhas = arquivo.readlines()

In [10]: 1 linhas
Out[10]: ['Batatinha quando nasce espalha a rama pelo chão.\n',
'Menininha quando dorme põe a mão no coração.\n',
'Sou pequeninha do tamanho de um botão.\n',
'Carrego papai no bolso e mamãe no coração.\n',
'O bolso furou e o papai caiu no chão.\n',
'Mamãe que é mais querida ficou no coração.\n']

In [11]: 1 for linha in linhas:
2         print(linha)

Batatinha quando nasce espalha a rama pelo chão.

Menininha quando dorme põe a mão no coração.

Sou pequeninha do tamanho de um botão,

Carrego papai no bolso e mamãe no coração

O bolso furou e o papai caiu no chão.

Mamãe que é mais querida ficou no coração.
```

Leitura em Modo Texto

- Caso queira utilizar o método `readline()`, o fim de arquivo é dado pela string varia (' ')

```
In [55]: 1 arquivo = open('batatinha.txt','r')
```

```
In [56]: 1 linha = arquivo.readline()
2 while linha != '':
3     print(linha)
4     linha = arquivo.readline()
5 arquivo.close()
```

Batatinha quando nasce espalha a rama pelo chão.

Menininha quando dorme põe a mão no coração.

Sou pequeninha do tamanho de um botão,

Carrego papai no bolso e mamãe no coração

O bolso furou e o papai caiu no chão.

Mamãe que é mais querida ficou no coração.

Leitura em Modo Texto

- A função `read()` irá ler tudo de uma vez

```
In [55]: arquivo = open('batatinha.txt', 'r')
```

```
In [56]: texto = arquivo.read()
```

```
In [57]: arquivo.close()
```

```
In [58]: texto
```

```
Out[58]: 'Batatinha quando nasce espalha a rama pelo chão.\nMenininha quando dorme põe a mão no coração.\nSou pequenininha do tamanho de um botão, \nCarrego papai no bolso e mamãe no c  
oração\nO bolso furou e o papai caiu no chão.\nMamãe que é mais querida ficou no coração.\n'
```

OBSERVAÇÃO: pode-se utilizar a função `strip()` da classe `str` para eliminar os `'\n'` do final da *string* lida

Leitura em Modo Texto

- O Python possui uma sintaxe especial (instrução `with`) para abrir o arquivo e já fechá-lo mediante a execução normal ou erro de leitura

```
In [60]: with open('batatinha.txt') as arquivo:
          for linha in arquivo:
              print(linha.strip())
```

Batatinha quando nasce espalha a rama pelo chão.
Menininha quando dorme põe a mão no coração.
Sou pequeninha do tamanho de um botão,
Carrego papai no bolso e mamãe no coração
O bolso furou e o papai caiu no chão.
Mamãe que é mais querida ficou no coração.

```
In [61]: with open('batatinha.txt') as arquivo:
          texto = arquivo.read()
          print(texto)
```

Batatinha quando nasce espalha a rama pelo chão.
Menininha quando dorme põe a mão no coração.
Sou pequeninha do tamanho de um botão,
Carrego papai no bolso e mamãe no coração
O bolso furou e o papai caiu no chão.
Mamãe que é mais querida ficou no coração.

Gravação em Modo Texto

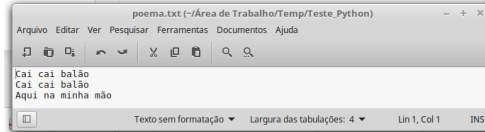
- Para gravar basta utilizar a função
 - `write()`, para gravar cada linha individualmente
 - `writelines()`, para gravar uma “lista de linhas”

```
In [12]: 1 arquivo.close()

In [16]: 1 arquivo = open('poema.txt','w')

In [17]: 1 arquivo.write('Cai cai balão\n')
          2 arquivo.write('Cai cai balão\n')
          3 arquivo.write('Aqui na minha mão\n')
Out[17]: 18

In [18]: 1 arquivo.close()
```



The screenshot shows a text editor window titled 'poema.txt (~/Área de Trabalho/Temp/Teste_Python)'. The menu bar includes 'Arquivo', 'Editar', 'Ver', 'Pesquisar', 'Ferramentas', 'Documentos', and 'Ajuda'. The toolbar contains icons for file operations and search. The text area displays the following content:

```
Cai cai balão
Cai cai balão
Aqui na minha mão
```

The status bar at the bottom indicates 'Texto sem formatação', 'Largura das tabulações: 4', 'Lin 1, Col 1', and 'INS'.

Tratamento de Exceções

- Durante a execução do programa, podem ocorrer alguns fatos podem gerar um erro e interromper a execução do seu programa \Rightarrow exceções
- Ao gerar a exceção, além de ser exibida uma mensagem nada amigável, a execução do programa é interrompida
- Para tratar esses quesitos mencionados acima, é necessário realizar o tratamento de exceções

Tratamento de Exceções

- Sintaxe:

```
1  """todo o conteúdo passível de gerar uma exceção
2  e que deseja tratar deve estar dentro de um
3  bloco try"""
4  try:
5      [codigo]
6  """O bloco except conterá o tratamento de um
7  tipo de exceção. Caso não seja especificado
8  o tipo de exceção, será tratado qualquer
9  tipo de exceção no bloco"""
10 except [tipo_excecao_0] as [identificador_excecao]:
11     [codigo]
12 except [tipo_excecao_2] as [identificador_excecao]:
13     [codigo]
14 """O bloco finally é opcional e será executado
15 tanto se houver quanto se não houver uma
16 exceção"""
17 finally:
18     [codigo]
```

- OBSERVAÇÃO:** é possível criar um único bloco except para tratar mais de um tipo de exceção \Rightarrow as múltiplas exceções devem ser separadas por vírgula

Tratamento de Exceções

- Criando um tratamento de exceções para arquivo não localizado

```
1 arquivo = open('batatinha2.txt', 'r')
```

```
-----  
FileNotFoundError                                Traceback (most recent call last)  
<ipython-input-57-9401254c5d46> in <module>  
----> 1 arquivo = open('batatinha2.txt', 'r')  
  
FileNotFoundError: [Errno 2] No such file or directory:  
y: 'batatinha2.txt'
```

Tratamento de Exceções

- Criando um tratamento de exceções para arquivo não localizado

```
1 arquivo = None
2 try:
3     arquivo = open('batatinha2.txt', 'r')
4     linhas = arquivo.readlines()
5     for linha in linhas:
6         print(linha)
7 except FileNotFoundError as erro:
8     print('Arquivo não localizado')
9     print('Mensagem do erro:', erro.strerror)
10 except:
11     print('Houve um erro na execução')
12 finally:
13     if arquivo != None:
14         arquivo.close()
```

Arquivo não localizado
Mensagem do erro: No such file or directory

Material Complementar

- Importar arquivo de outra pasta subindo um nível

https://groups.google.com/forum/#!topic/python-brasil/G12_1RRTZuQ

- Capítulo 6 - Arquivos

<https://www.caelum.com.br/apostila-python-orientacao-objetos/arquivos-e-modulos/>

- Capítulo 12 - Exceções e Erros

<https://www.caelum.com.br/apostila-python-orientacao-objetos/excecoes-e-erros/>

Material Complementar

- Capítulo 5 - Funções

`https:`

`//www.caelum.com.br/apostila-python-orientacao-objetos/funcoes/`

- Capítulo 7 - Orientação a Objetos

`https://www.caelum.com.br/apostila-python-orientacao-objetos/`
`orientacao-a-objetos/`

- Capítulo 11 - Herança Múltipla e Interfaces

`https://www.caelum.com.br/apostila-python-orientacao-objetos/`
`heranca-multippla-e-interfaces/`

- Manipulando Strings com Python

`https://wiki.python.org.br/ManipulandoStringsComPython`

Imagem do Dia

* Pais:

* Estado:

* Empresa (opcional):

☒ Salve este endereço em meu livro de endereços

O campo 'Empresa (opcional)' é obrigatório.

OK

Etapa 3: Detalhes de entrega

Etapa 4: Método de entrega

Tópicos em Inteligência Artificial

<http://ava.ufms.br/>

Rafael Geraldeli Rossi
rafael.g.rossi@ufms.br