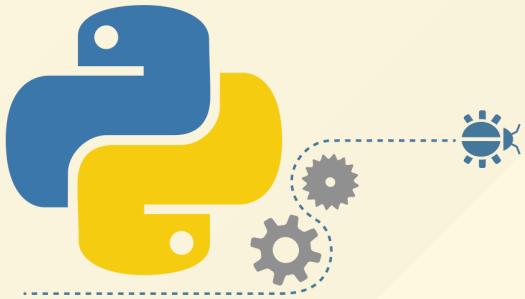

Python Fundamentals

Módulo 3

Clodonil Trigo (@clodonil)

Principais Métodos - Strings

```
s = "se nada mudar, invente."
```

Método	Sintaxe	Exemplo
s.capitalize()	s.capitalize()	"Titulo"
s.count(t, end)	`s.count('e')` e `s.count('e',2,23)`	3 e 2
s.find(t, start, end)	`s.find('e')` e `s.find('e',2,20)`	1, 18
s.join(t)	`s = '.'` , `t='dois'` e `s.join(t)`	'd.o.i.s'
s.lower()	`s.lowe()`	
s.upper()	`s.upper()`	
s.split(t, n)	`s.split('e')`	
s.title()	`s.title()`	'Se Nada..."

Principais Métodos - List

```
lista=[]
```

Méthodo	Descrição	Exemplo
append	Adiciona um item no final da lista	`lista.append('jose')`
copy	Faz uma copia da lista	`lista_b = lista.copy()`
count	retorna o número de ocorrência de um item	`lista.count('jose')`
extend	Prolonga a lista, adicionando no fim todos os elementos de outra lista	`lista.extend([10,'maria'])`
index	Retorna o índice do primeiro item pesquisado	`lista.index('jose')`
insert	Insere um item em uma posição especificada.	`lista.insert(0,'pedro')`

Principais Métodos - List

Méthodo	Descrição	Exemplo
pop	Remove o item na posição dada.	`lista.pop()` `lista.pop(0)`
remove	Remove o primeiro item encontrado na lista conforme o valor passado.	`lista.remove('jose')`
reverse	Inverte a ordem dos elementos na lista	`lista.reverse()`
sort	Ordena os itens na própria lista	`lista.sort()`
clear	Limpa toda a lista (remove tudo)	`lista.clear()`

Principais Métodos - Dictionaries

```
st = {'SP':'São Paulo','RJ':'Rio de Janeiro','MG':'Minas Gerais' }
```

Métodos	Descrição	Exemplo
st.copy()	Faz uma cópia do dicionário	`new = st.copy()`
st.get()	Retorna um valor de uma chave	`st.get('SP')` , `st['SP']`
st.items()	Retorna uma visão do dicionário (key e value)	`st.items()`
st.keys()	Retorna uma visão do dicionário (key)	`st.keys()`
st.values()	Retorna uma visão do dicionário (value)	`st.values()`
st.pop()	Remove e Retorna um valor de uma chave.	`st.pop('RS')`
st.update()	Atualiza um dicionário.	`st.update({'RS':'Rio Grande do Sul'})`

Módulo 3

Paradigma de Programação

MultiParadigma

Como todas as linguagens modernas, `Python é multiParadigma`. Isso significa que pode ser programada em mais de um `estilo`.

Programação Imperativa

> Na programação imperativa, o foco está no ato de `mudar variáveis`.

```
string = 'Python'
lista = [] # estado inicial

for l in string:
    lista.append(l) # cada iteração gera um novo estado

print(lista) # ['P', 'y', 't', 'h', 'o', 'n']
```

O fundamento da programação imperativa é o conceito de Máquina de Turing.

Paradigma Funcional

> Trata a computação como uma avaliação de funções matemáticas e que evita estados ou dados mutáveis. Ela enfatiza a aplicação de funções.

```
string = lambda x: x
lista = list(map(str, string('Python')))
print(lista) # ['P', 'y', 't', 'h', 'o', 'n']
```

Estruturada (Procedural)

> Estrutura a execução através de rotinas que são chamadas conforme a necessidade. Normalmente as chamamos de procedures/funções.

```
def soma(x,y):
    result = x + y
    return result


if __name__ == "__main__":
    x= soma(10,30)
    print(x)
```

“ No desenvolvimento procedural, é muito criar `biblioteca de funções` para serem utilizados em vários projetos diferentes. ”

import

> Das bibliotecas de função, pode ser importado o arquivo inteiro, com todas as funções ou apenas funções especificas.

- Importar todas as funções:

```
import calc

calc.soma(10,20)
```

- Importar funções especificas:

```
from calc import soma

soma(10,20)
```

Escopo de Variável

Toda variável declarada fora de uma função, são do tipo `global` e podem ser acessadas pelas funções, mais `não podem ser modificadas` diretamente.

Exemplo de acesso:

```
x = 30
y = 20
controle = 0

def verifica():
    if x > y:
       print('maior')
    else:
       print('menor')

verifica()
```

Exemplo Alteração:

Erro na alteração

```
x = 30
y = 20
controle = 0
def verifica():
    if x > y:
        print('maior')
        controle += x
    else:
        print('menor')


verifica()
```

Exemplo Alteração:

Acessando variável Global

```
x = 30
y = 20
controle = 0
def verifica():
    global controle
    if x > y:
       print('maior')
       controle += x
    else:
       print('menor')


verifica()
```

Desenvolver as procedures para implementar uma movimentação e conta bancária.

- **Procedures**:
  - criar_conta
  - deposito
  - saque
  - extrato
- **Atributos da Conta**:
  - Numero da Conta
  - titular
  - Saldo
  - Limite

- File: banco.py

```
def criar_conta(nconta, titular, saldo, limite):
    conta = {"nconta": nconta,
             "titular": titular,
             "saldo": saldo,
             "limite": limite}
    return conta
```

Validando a criação do método:

```
>>> from banco import criar_conta
>>> clodonil = criar_conta(123,'clodonil',100,500)
>>> clodonil['nconta']
123
```

- Agora vamos implementar a procedure para realizar o depósito na conta.

```
def deposito(conta, valor):
    conta ["saldo"] += valor
```

- Da mesma forma vamos implementar o módulo saque.

```
def saque(conta,valor):
    conta["saldo"] -= valor
```

- E por último o extrato

```
def extrato(conta):
    print("Sr(a) {0} o Saldo é {1}".format(conta["titular"],conta["saldo"]))
```

file: conta.py

Importar as procedures/funções:

```
from banco import criar_conta, deposito, saque, extrato

if __name__ == "__main__":
    conta_do_jose = criar_conta(3123,'Jose',50,1500)
    deposito(conta_do_jose, 100)
    extrato(conta_do_jose)
    saque(conta_do_jose, 50)
```

Orientação a Objeto

O Paradigma orientação a objetos(OO), tem como proposta resolver problemas atráves de troca de mensagens entre Objetos, que são representações dos problemas do cotidiano.

Classes

- O método especial `__init__()` representa o construtor da classe e será executada somente quando a classe é criada;
- `Self` representa o endereçamento do objeto na mémoria, utilizado para acessar os atributos e métodos.

banco.py

```
class Conta:
  def __init__(self,nconta, titular, saldo, limite):
    print("Construindo objeto...")
    self.nconta = nconta
    self.titular = titular
    self.saldo = saldo
    self.limite = limite

  def deposito(self, valor):
    self.saldo += valor

  def saque(self, valor):
    self.saldo -= valor

  def extrato(self):
    print("Sr(a) {0} o Saldo eh {1}".format(self.titular,self.saldo))
```

Agora que temos a classe `Conta`, podemos criar vários objetos do tipo conta:

```
>> from banco import Conta
>>> x = Conta(12321,'Clodonil',100,200)
Construindo objeto...
>>> x
<banco.Conta object at 0x7f1ba4093d10>
>>> x.saldo
100
>>> x.deposito(100)
>>> x.saldo
200
>>>
```

Encapsulamento

podemos deixar o atributo no modo `privado`, e para isso adicionamos __ (underscore) antes do nome da variável.

```
class Conta:
   def __init__(self,nconta, titular, saldo, limite):
     print("Construindo objeto...")
     self.__nconta = nconta
     self.__titular = titular
     self.__saldo = saldo
     self.__limite = limite
```

Métodos para implementar os getters e setter para ter acesso ao atributo protegido.

conta.py

```
@property
def saldo(self):
    return self.__saldo

@property
def titular(self):
    return self.__titular.capitalize()

@property
def limite(self):
    return self.__limite

@limite.setter
def limite(self, limite):
    self.__limite = limite
```

Herança

- A Herança é um conceito do paradigma da orientação à objetos que determina que uma classe pode `herdar atributos e métodos` de uma outra classe e, assim, evitar que haja muita `repetição de código`.
- Utilizando o exemplo da conta, vamos criar a classe de `Investimento`, herdando todos os métodos e atributos da classe `Conta`.

Herança

Criando a classe `Investimento` e herdando os métodos e atributos na classe `Conta`.

```
class Investimento(Conta):
    pass
```

- Validando a criação da classe `Investimento`.

```
>>> from conta import Conta, Investimento
>>> maria = Investimento(1321,'Maria',100,200)
Construindo objeto...
>>> maria.titular
'Maria'
>>> maria.limite
200
```

Sobrecarga de Métodos

Muitas vezes precisamos rescrever um método que foi implementado na classe `Pai`. Um exemplo muito comum é o método `__init__()`. Para instânciar o método da classe `Pai`, usamos o comando `super()`.

```
class Investimento(Conta):
  def __init__(self, nconta,titular, saldo, limite,juros):
    super().__init__(nconta, titular, saldo, limite)
    self.__juros = juros
```

Para finalizar, vamos implementar o método rendimento. O método rendimento utilizar a variável `saldo` que está privada na classe `Conta`. Precisamos mudar esse atributo para `(_) underscore`.

conta.py

```
class Investimento(Conta):
    def __init__(self, nconta,titular, saldo, limite, juros):
      super().__init__(nconta, titular, saldo, limite)
      self.__juros = juros

    def redimento(self):
        self._saldo += (self._saldo * self.__juros)
        return self._saldo
```

Laboratório Módulo 3