

Repositório para salvar códigos de PLP

REOs

- [REO 2 - Paradigma Imperativo](#)
- [REO 3 - Paradigma Orientado a Objetos](#)
- [REO 4 - Paradigma Funcional](#)

Reo 2 - Paradigma Imperativo

* [Configurações](#reo2Configurações) * [Ambiente virtual](#ambienteVirtualPython) * [Extensões para VsCode](#extensoesParaVsCodePython) * [settings.json](#settings.jsonPython) * [Aulas](#reo2Aulas) * [Videoaula de introdução ao Python: GCC198](#aula01) * [Paradigma Imperativo : Variáveis e Tipos de Dados](#aula02) * [Paradigma Imperativo : Avaliação de Expressões e Controle de Fluxo](#aula03) * [Paradigma Imperativo : Subprogramas](#aula04) * [Atividade Avaliativa](#reo2AtividadeAvaliativa)

Configurações

- Ambiente Virtual
 - Windows

```
py -3 -m venv venv
//NO VENV
pip install wheel
```

- Linux e Mac

```
python3 -m venv venv
//NO VENV
venv/bin/activate
pip install wheel
deactivate
```

- Extensões para VSCode:
 - Code Runner
 - Python (Microsoft)
 - Python Test Explorer for Visual Studio Code
 - Python Preview
 - Python Docstring Generator
- settings.json

Na pasta do projeto, crie uma pasta chamada .vscode e dentro dela um arquivo chamado settings.json

- Windows

```
{  
    "python.pythonPath": "venv\\Scripts\\python.exe",  
    "code-runner.executorMap": {  
        "python": "venv\\Scripts\\python.exe",  
    },  
    "code-runner.ignoreSelection": true,  
    "code-runner.runInTerminal": true,  
    "python.linting.mypyEnabled": true,  
    "python.linting.flake8Enabled": true,  
    "python.testing.unittestEnabled": true,  
    "[python)": {  
        "editor.formatOnSave": true  
    }  
}
```

- Linux

```
{  
    "python.pythonPath": "venv/bin/python",  
    "code-runner.executorMap": {  
        "python": "venv/bin/python",  
    },  
    "code-runner.ignoreSelection": true,  
    "code-runner.runInTerminal": true,  
    "python.linting.mypyEnabled": true,  
    "python.linting.flake8Enabled": true,  
    "python.testing.unittestEnabled": true,  
    "[python)": {  
        "editor.formatOnSave": true  
    }  
}
```

Aulas

- **Videoaula de introdução ao Python: GCC198**

- [Conditionals](#)
- [Iterations](#)
- [Exceptions](#)
- [Files](#)
- [CommandArguments](#)
- [Extras](#)

-
- **Conditionals**

■ Vídeo-aula de introdução ao Python : GCC198

```
valor = input("Digite um valor")
valor = int(valor)
if ((valor % 2) == 0):
    print('Número par')
else:
    print('Número ímpar')
```

Digite um número 2

Número par

```
valor = int(input("Digite um valor"))

if valor == 0:
    print('Zero!')
elif valor % 2 == 0:
    print('Número par')
else:
    print('Número ímpar')
```

Digite um número 0

Zero!

```
valor = int(input("Digite um valor"))

msg = 'par' if valor % 2 == 0 else 'ímpar'
print(msg)
```

Digite um número 2

par

◦ **Iterations**

```
for i in range(10):
    print(i)
```

0

1

2

```
3  
4  
5  
6  
7  
8  
9
```

```
for i in range(5,10):  
    print(i)
```

```
5  
6  
7  
8  
9
```

```
blacklist = ["palavrão","palavrona","palavreadão"]  
for palavra in blacklist:  
    print("Palavra proibida: {}".format(palavra))
```

```
Palavra proibida: palavrão  
Palavra proibida: palavrona  
Palavra proibida: pavreadão
```

```
blacklist = ["palavrão","palavrona","palavreadão"]  
texto = input("Digite uma frase: ")  
palavras = texto.split()  
for palavra in palavras:  
    if palavra.lower() in blacklist:  
        print("A palavra {} é proibida!".format(palavra))
```

```
Digite uma frase: Oi, palavrão tal  
A palavra palavrão é proibida!
```

- **Exceptions**

```

try:
    valor = int(input("Digite um número"))
    if valor == 0:
        print('Zero!')
    elif valor % 2 == 0:
        print('Número par')
    else:
        print('Número ímpar')
except:
    print('Valor digitado não é um número!')

```

Digite um número a

Valor digitado não é um número!

- o **Files**

dataset.csv :

7,8,9

3,4,5

2,4,1

90,89,20

8,4,12

```

xs = []
ys = []
zs = []
with open('dataset.csv','r') as file:
    lines = file.readlines()
    for line in lines:
        x, y, z = line.split(',')
        xs.append(x)
        ys.append(y)
        zs.append(z)

print(xs)
print(ys)
print(zs)

```

['7', '3', '2', '90', '8']

['8', '4', '4', '89', '4']

['9\n', '5\n', '1\n', '20\n', '12']

```

xs = []
ys = []
zs = []
with open('dataset.csv','r') as file:
    lines = file.readlines()
    for line in lines:
        x, y, z = line.split(',')
        xs.append(x)
        ys.append(y)
        zs.append(z.strip())

print(xs)
print(ys)
print(zs)

```

['7', '3', '2', '90', '8']

['8', '4', '4', '89', '4']

['9', '5', '1', '20', '12']

- **Command Arguments**

- python_example1.py

```

import sys

caminho_do_arquivo = sys.argv[1] #o sys.argv[0] é o nome do
#próprio arquivo python

xs = []
ys = []
zs = []

with open(caminho_do_arquivo, 'r') as file:
    lines = file.readlines()
    for line in lines:
        x, y, z = line.split(',')
        xs.append(x)
        ys.append(y)
        zs.append(z.strip())

print(xs)
print(ys)
print(zs)

```

- A chamada é feita por terminal de comando

python3 python_example1.py dataset.csv

```
[7', '3', '2', '90', '8']  
[8', '4', '4', '89', '4']  
[9', '5', '1', '20', '12']
```

- **Extras**

```
numeros = [1, 10, 100, 1000, 2, 20, 200]  
print(sum(numeros))  
print(max(numeros))  
print(min(numeros))
```

```
1333  
1000  
1
```

```
numeros = [1, 10, 100, 1000, 2, 20, 200]  
print("Média da lista", sum(numeros)/len(numeros))
```

```
190.42857142857142
```

- **Paradigma Imperativo : Variáveis e Tipos de Dados**

- [Vídeo-aula Paradigma Imperativo : Variáveis e Tipos de Dados](#)
- [Slide Variáveis e tipos de dados](#)

```
my_string = 'Hello, World!'  
my_flt = 45.06  
my_bool = 5 > 9  
my_list = ['item1','item2']  
my_tuple = ('item1','item2')  
my_dict = {'letter':'g','number':7}
```

- **Paradigma Imperativo : Avaliação de Expressões e Controle de Fluxo**

- [Paradigma Imperativo : Avaliação de Expressões e Controle de Fluxo](#)
- [Avaliações de expressões e controle de fluxo](#)

- **Paradigma Imperativo : Subprogramas**

- [Paradigma Imperativo : Subprogramas](#)
- [Subprogramas](#)

```
def subprograma(a,b,c):
    print(a + b + c)
```

- Recursao

```
def fatorial(n):
    if (n <= 1):
        return 1
    else:
        return (n * fatorial(n-1))
```

Atividade Avaliativa

- Sobre
 - Programa pra rodar o coeficiente da correlação de Pearson
- Rodar

```
py -3 atividade_avaliativa_Rafael_Porto_reo2 teste.txt
```

Reo 3 - Paradigma Orientado a Objetos

- Aulas
 - **Paradigma Orientado a Objetos : Conceitos iniciais**
 - **Paradigma Orientado a Objetos : Encapsulamento**
 - **Paradigma Orientado a Objetos : Herança e Composição**
 - **Paradigma Orientado a Objetos : Polimorfismo**
- Python
- Atividade Avaliativa

Paradigma Orientado a Objetos : Conceitos iniciais

[Video Aula](#)

Conjunto de princípios

- Orientam a criação de sistemas computacionais, objetos que interagem entre si.

Em termos de LPs, conceitos formais surgem com Simula 67, sendo consolidados com Smalltalk (primeira linguagem orientada a objetos).

Popularizado com a difusão de interfaces gráficas de usuários (GUIs)

- Surgimento de ferramentas com suporte para desenvolvimento de aplicações gráficas (C++, FoxPro, Delphi).

Suportado por várias linguagens (ex: Python, Ruby, C#)

- Atualmente sua maior expressão comercial é dada pelo Java

Pilares da OO

Conceitos fundamentais (pilares) que norteiam o desenvolvimento OO:

- Abstração;
- Encapsulamento;
- Herança;
- Polimorfismo.

Abstração

Representação de uma entidade do mundo real, com seu comportamento e características.

"Modelos Mentais"

- Classes;
- Objetos;
- Métodos;
- Atributos;

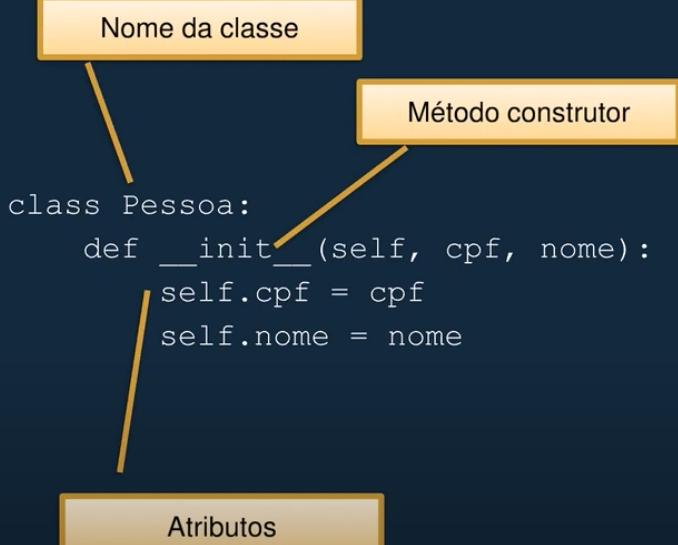
Classes: Uma classe pode ser entendida como um módulo ou uma estrutura de dados abstrata.

Uma visão mais ampla pode levar à seguinte definição:

- Uma classe é um tipo abstrato de dados, que reúne objetos com características similares.
- O comportamento destes objetos é descrito pelo conjunto de métodos disponíveis.
- O conjunto de atributos da classe descrevem as características de um objeto.

Classes

Considere a definição de uma classe Pessoa. Existem diversas pessoas, e cada um deles se diferencia pelo nome e cpf.



Objetos:

Um objeto pode ser entendido como um ser, lugar, evento, coisa ou conceito do mundo real que possa ser aplicável a um sistema.

É comum que haja objetos diferentes com características semelhantes. Esses objetos são agrupados em classes.

Classes são um agrupamento de objetos com características similares!

Objetos são entidades (instâncias) únicas de uma classe!

Objetos

Considere a classe Pessoa podemos ter um objeto p1, com cpf 123.456.789-10 e nome João da Silva.

variável que armazena
objeto criado

Instanciação (criação) de um objeto da
classe Pessoa

```
p1 = Pessoa('123.456.789-10', 'João da  
Silva')
```

Atributos:

Um atributo é uma característica de um grupo de entidades do mundo real, agrupados em uma classe.

Um atributo pode ser um valor simples (um inteiro, por exemplo) ou estruturas complexas (um outro objeto, por exemplo).

Atributos

Considere a classe Pessoa. Seus atributos são o cpf e o nome.

```
class Pessoa:  
    def __init__(self, cpf, nome):  
        self.cpf = cpf  
        self.nome = nome
```

Atributos

Atributos de classe

- Em geral, os atributos pertencem a cada objeto instanciado, ou seja, a cada novo instanciação de uma mesma classe, cada instância pode ter valores distintos para cada atributo.
- Atributos de classe são definidos para terem o mesmo valor para todas as instâncias de uma classe.

Exemplo: Atributos de classe

```
class Pessoa:  
    __total_pessoas = 0  
  
    def __init__(self, cpf, nome):  
        self.cpf = cpf  
        self.nome = nome  
        Pessoa.__total_pessoas += 1  
    def get_total_pessoas(self):  
        return Pessoa.__total_pessoas  
  
p1 = Pessoa('123.456.789-10', 'Fulano Ciclano', '01/02/1995')  
print(p1.get_total_pessoas()) # erro  
print(Pessoa.get_total_pessoas(p1)) # OK
```

Atributos de instância. Esses atributos receberão valores distintos a cada instanciação

Em Python, atributos de classe devem ser definidos com dois 'underlines' como prefixo do nome da variável

Forma de acessar atributo de classe

Métodos

Semelhante a uma função, é a implementação de uma ação da entidade representada pela classe; Conjunto de métodos define o comportamento dos objetos de uma classe.

Métodos

```
from datetime import datetime

class Pessoa:
    def __init__(self, cpf, nome, data_nascimento):
        d, m, a = data_nascimento.split("/")
        self.cpf = cpf
        self.nome = nome
        self.data_nascimento = datetime(a, m, d)

    def get_data_nascimento(self):
        return self.data_nascimento.strftime("%x")
```

Método que retorna a data de nascimento de uma pessoa. O `self.` indica que o atributo pertence ao objeto.
Semelhante ao `this.` do Java

Construtores

É um método especial para a criação e inicialização de uma nova instância de uma classe.

Um construtor inicializa um objeto e suas variáveis, cria quaisquer outros objetos de que ele precise, garantindo que ele seja configurado corretamente quando criado.

Na maioria das LPs, o construtor é um método que tem o mesmo nome da classe, que geralmente é chamado quando um objeto da classe é declarado ou instanciado.

Exemplo: Construtores

```
from datetime import datetime
```

```
class Pessoa:
```

```
    def __init__(self, cpf, nome, data_nascimento):  
        d, m, a = map(int, data_nascimento.split("/"))  
        self.cpf = cpf  
        self.nome = nome  
        self.data_nascimento = datetime(a, m, d)
```

Esse construtor recebe 3 parâmetros reais, o primeiro (não contado aqui), indicado como `self`, serve como referência para o próprio objeto.

Exemplo: Construtores

Imagine o caso de um prontuário médico, poderíamos ter, adicionalmente, as seguintes classes:

```
class Medicamento:
```

```
    def __init__(self, nome):  
        self.nome = nome
```

```
class Prontuario:
```

```
    def __init__(obj, paciente):  
        obj.paciente = paciente  
        obj.medicamentos = []
```

Observe que, ao invés de usar `self`, foi utilizado `obj` como referência ao objeto. Não se trata de uma palavra reservada, mas do primeiro argumento de um método do objeto.

Costuma-se utilizar `self`.

Destruidores

De forma similar aos construtores, os destruidores são métodos fundamentais das classes, sendo geralmente chamados quando termina o tempo de vida do objeto.

Em algumas linguagens como C++, ocupam um papel tão importante quanto os construtores, por conta da necessidade de desalocação de memória.

Em outras linguagens como Java, o Garbage Collector (Coletor Automático de Lixo) faz esse papel, deslocando aquilo que não é mais utilizado. Há o método `finalize()`, mas raramente é utilizado (há dúvidas se sempre funciona, inclusive).

Tanto os construtores, quanto os destruidores são métodos que não precisam ser definidos em Orientação a Objetos em Python, caso o comportamento esperado seja o padrão.

Geralmente, define-se o construtor para a passagem de argumentos na criação do objeto. Já o destrutor não se costuma definir.

Caso seja necessário realizar algum procedimento na destruição do objeto, define-se o método destrutor, como será exemplificado.

Destruidores

```
p1 = Pessoa('123.456.789-10', 'Fulano Ciclano', '01/02/1995')  
p2 = Pessoa('123.456.789-11', 'Ciclano Fulano', '31/12/1996')
```

```
del p1 # nesse caso, o objeto instanciado é desalocado da memória (destruído)  
del p2 # nesse caso, o objeto instanciado é desalocado da memória (destruído)
```

Destruidores

```
class A:
```

```
    def __init__(self):  
        print ("A has been created")
```

```
    def __del__(self):  
        print ("A has been destroyed")
```

Método destrutor dos objetos da classe A.

```
a = A()  
del a
```

Quando o objeto a for destruído, a mensagem definida no destrutor será exibida

Garbage Collection em Java

- Em C++ a memória é alocada e desalocada explicitamente
- Java possui gerenciamento automático de memória, realizado pela JVM
 - Evita vazamento de memória e bugs de ponteiros
 - Consome recursos computacionais quanto à decisão de desalocação
 - É um processo "não determinístico"

Paradigma Orientado a Objetos : Encapsulamento

[Video Aula](#)

Na programação Orientada a Objetos, é desejável e, muitas vezes, muito importante, que os atributos dos objetos tenham o devido nível e forma de acesso externo ao objeto.

Para isso, é necessário definir a visibilidade dos atributos e métodos de um objeto.

Como 'dono' dos atributos, um objeto é o mais indicado para lidar com seus atributos e métodos e não o cenário externo, como outros objetos.

O encapsulamento permite maior controle e validação dos dados de um objeto

Encapsulamento

É importante evitar que atributos de uma classe sejam diretamente acessíveis de fora da classe.

```
class Conta:  
    def __init__(self):  
        self.saldo = 0  
  
c1 = Conta()  
c1.saldo = 100000
```

O atributo saldo, de um objeto da classe Conta poderá ter esse valor acessado externamente. O que pode levar a erros na validação dos dados. Como evitar isso?

```
class Conta:  
    def __init__(self):  
        self.saldo = 0  
  
c1 = Conta()  
c1.saldo = 100000000
```

Para acessar esses atributos, métodos são definidos

- permitem maior controle dos valores, como validação dos dados

Visibilidade

A visibilidade é utilizada para indicar o nível de acesso de um determinado atributo ou método;

Os três modos distintos são:

- Público:
 - Objetos de quaisquer classes podem ter acesso a atributos, ou métodos, públicos;
- Privado:
 - Apenas a classe que define atributos ou métodos privados pode ter acesso a eles;
- Protegido:
 - Apenas a classe e suas subclasses podem ter acesso a atributos e métodos protegidos;

Atributos privados em Python

class Conta:

A definição de um atributo privado é feita com o uso de dois *underscores* antes do nome do atributo. Equivale a **private** em Java e C++

```
__slots__=[__saldo]  
def __init__(self):  
    self.__saldo = 0
```

Assim, um atributo privado não pode ser acessado (teoricamente) diretamente, de fora do objeto

```
c1 = Conta()  
c1.saldo = 100000000 # erro
```

Visibilidade de atributos

class A():

```
def __init__(self):
```

```
    self.__priv = "I am private"  
    self._prot = "I am protected"  
    self.pub = "I am public"
```

Dois *underscores* indicam atributo privado. Um *underscore*, protegido. Nenhum, público

Em Python é possível definir atributos, no momento da execução:

Definindo e limitando atributos

O uso de `__slots__` faz a restrição dos atributos do objeto, não permitindo a criação de novos atributos

```
class Conta:  
    __slots__ = ('_Conta__saldo')  
    def __init__(self):  
        self.__saldo = 0  
    def set_saldo(self, saldo):  
        if saldo < 0:  
            raise Exception('Error')  
        self.__saldo = saldo
```

Não é permitida a criação de novos atributos nos objetos da classe Conta

```
c1 = Conta()  
c1.new_saldo = 100000000 # Erro
```

Definindo e limitando atributos

O uso de `__slots__` faz a restrição dos atributos do objeto, não permitindo a criação de novos atributos

```
class Conta:  
    __slots__ = ('_Conta__saldo')  
    def __init__(self):  
        self.__saldo = 0  
    def set_saldo(self, saldo):  
        if saldo < 0:  
            raise Exception('Error')  
        self.__saldo = saldo
```

É uma forma de acessar atributos privados! Python é uma linguagem muito dinâmica, 'pra gente adulta que sabe o que faz'. Mas tem como evitar?

```
c1 = Conta()  
c1.new_saldo = 100000000 # Erro  
c1._Conta__saldo = -100000000 # permitido
```

Getters e Setters

São métodos específicos para acesso aos atributos de uma classe, principalmente os atributos privado.

Como padrão na comunidade de programadores, são nomeados com os prefixos 'ser_' ou 'get_' para ajustar ou obter os valores dos atributos.

Permitem validação e formatação dos valores dos atributos antes de serem acessados ou alterados fora do objeto.

São métodos, geralmente, públicos

Troca de Mensagens

Na Orientação a Objetos, os objetos interagem pela troca de mensagens, e, nesse contexto, os métodos getters e setters desempenham papel importante e frequente.

Cada objeto sabe os atributos que têm e, portanto, têm métodos para alterá-los adequadamente

Padrões de Projeto de Software

São soluções gerais para problemas que ocorrem com frequência na programação.

Um desses padrões é chamado '*Decorator*'

- Esse padrão adiciona comportamento a um método ou objeto em tempo de execução.

No python: **Decorators** @property e @attr.setter

@property decora os métodos getters

@attribute_name.setter, os métodos setters

Não se utiliza os prefixos 'get_' e 'set_'

Os métodos têm o nome do atributo a ser manipulado

- Polimorfismo

The diagram features a dark blue background with light blue hexagonal patterns. In the center, the title 'Decorators @property e @attr.setter' is displayed in white. Below the title, there is a code snippet for a 'Conta' class. Two callout boxes with orange borders point from the left side towards the code. The top box contains the text: 'Decorator @property. Transforma o método num getter do atributo saldo'. The bottom box contains the text: '@saldo.setter torna o método num setter'. Arrows from these boxes point to the corresponding code lines: the first arrow points to the @property decorator above the 'saldo' method, and the second points to the @saldo.setter decorator above the 'saldo' method definition.

```
class Conta:  
    __slots__ = ('_Conta__saldo')  
    def __init__(self):  
        self.__saldo = 0  
    @property  
    def saldo(self):  
        return R$ {0:.2f}'.format(self.__saldo)  
    @saldo.setter  
    def saldo(self, novo_saldo):  
        if novo_saldo > 0:  
            self.__saldo = novo_saldo  
        else:  
            raise Exception()  
  
>> c1 = Conta()  
>> c1.saldo = 100 # chama setter  
>> c1.saldo # chama getter  
R$ 100.00  
>> c1.saldo = -100 # Exception
```

Decorator

@classmethod define métodos de classe

@classmethod recebe uma referência à classe (geralmente chamado de cls) como primeiro parâmetro implícito (semelhante ao self, referência ao objeto)

Decorators @classmethod

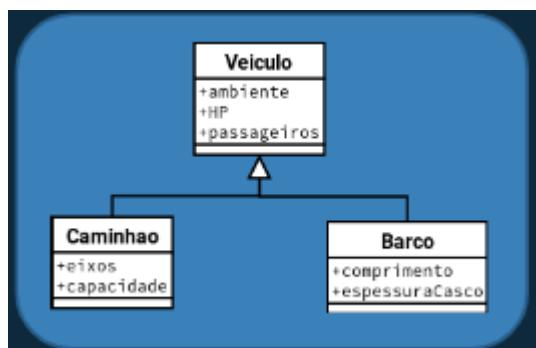
```
__contas é definida como uma variável privada, de classe.  
Decorator @classmethod.  
Transforma o método acessível a nível de classe e não de objeto.  
  
A cada novo objeto instanciado,  
__contas é incrementada. O método contas_instanciadas  
acessa a variável de classe e informa a quantidade de objetos  
instanciados.  
  
Pode-se acessar o método de classe tanto pela classe, quanto  
por um objeto instanciado
```

```
class Conta:  
    __contas = 0  
    def __init__(self):  
        Conta.__contas += 1  
  
    @classmethod  
    def contas_instanciadas(cls):  
        return '{} contas ativas'.format(cls.__contas)  
  
>> c1 = Conta()  
>> print(Conta.contas_instanciadas())  
1 contas ativas  
>> c2 = Conta()  
>> print(c2.contas_instanciadas())  
2 contas ativas
```

Paradigma Orientado a Objetos : Herança e Composição

HERANÇA OO

Mecanismo que permite que características comuns a diversas classes sejam organizadas em uma classe base e que, a partir dessa, outras possam ser criadas, herdando a classe base.



A classe derivada (ou subclasse) mantém as características herdadas e acrescenta o que for de sua exclusividade.

```
Veja o diagrama de classes do slide anterior.  
Exemplo em Java
```

```
public class Barco extends Veiculo {  
    private float comprimento;  
    private float espessuraCasco;  
  
    public Barco (String a, int hp, int p, float c, float e){  
        super(a, hp, p);  
        this.comprimento = c;  
        this.espessuraCasco = e;  
    }  
}
```

A classe derivada (ou subclasse) mantém as características herdadas e acrescenta o que for de sua exclusividade.

```

class Pessoa:
    def __init__(self, nome):
        self.nome = nome

class Paciente(Pessoa):
    def __init__(self, nome, med_id):
        super().__init__(nome)
        self.med_id = med_id

class Medico(Pessoa):
    def __init__(self, nome, id_func):
        super().__init__(nome)
        self.id_func = id_func

```

Classe base. Contém características comuns a qualquer pessoa.

Classe que herda a classe Pessoa. Adiciona atributos específico a um paciente

Classe que também herda a classe Pessoa. Adiciona atributos específico a um médico

class Pessoa:
**def __init__(self, nome):
 self.nome = nome**

class Paciente(Pessoa):
**def __init__(self, nome, med_id):
 super().__init__(nome)
 self.med_id = med_id**

class Medico(Pessoa):
**def __init__(self, nome, id_func):
 super().__init__(nome)
 self.id_func = id_func**

CLASSE ABSTRATA

Uma classe abstrata contém métodos abstratos, ou seja, que não têm implementação

As classes que herdarem a classe abstrata são obrigados a realizar a implementação dos métodos abstratos da classe abstrata

Uma classe abstrata, com métodos abstratos não pode ser diretamente instanciada

Módulo nativo do Python para lidar com classes e métodos abstratos

Classe abstrata, ela herda a classe 'abc.ABC'.

Método abstrato, definido pelo decorator `@abc.abstractmethod`. As classes que herdarem Pessoa, deverão implementar o método `definir_nome`

```
import abc
class Pessoa(abc.ABC):
    @abc.abstractmethod
    def definir_nome(self, nome):
        pass

pessoa1 = Pessoa() # erro: Can't instantiate abstract class Pessoa
with abstract methods definir_nome
```

Classe abstrata

Classe Paciente herda a classe abstrata Pessoa e implementa o método `definir_nome`

```
>> p1 = Paciente('João', 36348)
>> p1.nome
João
```

```
import abc
class Pessoa(abc.ABC):
    @abc.abstractmethod
    def definir_nome(self, nome):
        pass

class Paciente(Pessoa):
    def __init__(self, nome, med_id):
        self.med_id = med_id
        self.definir_nome(nome)
    def definir_nome(self, nome):
        self.nome = nome
```

DUCK TYPING

Estilo de codificação, em linguagens dinamicamente tipadas, em que define-se classes e métodos sem se importar com o tipo das variáveis.

Importa-se com o comportamento, não com o tipo

se anda como pato, nada como um pato e faz quack como um pato, então provavelmente é um pato

Por ser uma linguagem não tipada, ou seja, não se define o tipo das variáveis, os argumentos de métodos não são tipados e podem receber qualquer tipo de dados.

Obviamente, as expressões com tais argumentos devem envolver operadores que consigam lidar com os valores fornecidos.

Para se certificar que uma variável é um tipo esperado, o Python fornece algumas funções úteis:

- **`type()`** recebe como parâmetro uma variável e retorna o tipo da mesma
- **`isinstance()`** recebe dois parâmetros: variável e tipo esperado. Retorna True se a variável é do tipo indicado e False caso contrário

```
class A:
    pass

>> a = A()

>> isinstance(a,A)
True
```

```
>> type(a)
<class '__main__.A'>
```

HERANÇA MÚLTIPLA

Uma classe pode herdar de mais de uma classe seus atributos e métodos

Java não suporta

C++ e Python suportam herança múltipla

```
class Clock():
    def __init__(self, hours, minutes, seconds):
        self.set_clock(hours, minutes, seconds)

    def set_clock(self, hours, minutes, seconds):
        if type(hours) == int and 0 <= hours and hours < 24:
            self._hours = hours
        else:
            raise TypeError("Hours have to be integers between 0 and 23!")
        if type(minutes) == int and 0 <= minutes and minutes < 60:
            self._minutes = minutes
        else:
            raise TypeError("Minutes have to be integers between 0 and 59!")
        if type(seconds) == int and 0 <= seconds and seconds < 60:
            self._seconds = seconds
        else:
            raise TypeError("Seconds have to be integers between 0 and 59!")

    def __str__(self):
        return "{0:02d}:{1:02d}:{2:02d}".format(self._hours,
                                                self._minutes,
                                                self._seconds)

    def tick(self):
        if self._seconds == 59:
            self._seconds = 0
            if self._minutes == 59:
                self._minutes = 0
                if self._hours == 23:
                    self._hours = 0
                else:
                    self._hours += 1
            else:
                self._minutes += 1
        else:
            self._seconds += 1
```

```
class Calendar(object):
    months = (31,28,31,30,31,30,31,31,30,31,30,31)

    @staticmethod
    def leapyear(year):
        if not year % 4 == 0:
            return False
        elif not year % 100 == 0:
            return True
        elif not year % 400 == 0:
            return False
        else:
            return True

    def __init__(self, d, m, y):
        self.set_calendar(d,m,y)

    def set_calendar(self, d, m, y):
        self._days = d
        self._months = m
        self._years = y

    def __str__(self):
        return "{0:02d}/{1:02d}/{2:4d}".format(self._months, self._days, self._years)

    def advance(self):
        max_days = Calendar.months[self._months-1]
        if self._months == 2 and Calendar.leapyear(self._years):
            max_days += 1
        if self._days == max_days:
            self._days = 1
            if self._months == 12:
                self._months = 1
                self._years += 1
            else:
                self._months += 1
        else:
            self._days += 1
```

Classe *CalendarClock*

- Tem herança múltipla

Herança múltipla das classes
Clock e *Calendar*

Tem um método *tick*, que usa o
tick da classe *Clock* e o método
advance da classe *Calendar*

```
from clock import Clock
from calendar import Calendar

class CalendarClock(Clock, Calendar):
    def __init__(self, day, month, year, hour, minute, second):
        Clock.__init__(self, hour, minute, second)
        Calendar.__init__(self, day, month, year)

    def tick(self):
        previous_hour = self._hours
        Clock.tick(self)
        if (self._hours < previous_hour):
            self.advance()

    def __str__(self):
        return Calendar.__str__(self) + ", " + Clock.__str__(self)
```

PROBLEMA DO DIAMANTE

O problema do Diamante (devido à forma geométrica da ilustração ao lado) pode ocorrer na herança múltipla

```
class A:
    def m(self):
        print("m of A called")
class B(A):
    def m(self):
        print("m of B called")
class C(A):
    def m(self):
        print("m of C called")

class D(B,C):
    pass
```

Considere as classes A, B, C e D, a cima, o que acontece no seguinte trecho do código?

```
d = D()
d.m()
```

Qual método *m()* será invocado, da classe A,B ou C?

A resolução da ambiguidade depende da MRO (MethodResolutionOrder) de cada linguagem

Leia em <https://www.python.org/download/releases/2.3/mro/ADBC>

Paradigma Orientado a Objetos : Polimorfismo

[Video aula](#)

Habilidade de apresentar a mesma interface para formas diferentes, tipos diferentes de dados

- Por exemplo, métodos polimórficos podem aceitar tipos de dados diferentes e, dependendo dos mesmos ter comportamentos diferentes;

```

def f(x, y):
    print("values: ", x, y)

f(42, 43)
f(42, 43.7)
f(42.3, 43)
f(42.0, 43.9)
f([1,2,3], [6,5,4,3,4,7])
f({"A":65, "B":66}, {"C":67, "D":68})

```

O método `f` recebe dois parâmetros e simplesmente os exibe na tela. Por não ser uma linguagem tipada (Python). Esse método pode ser chamado com diversos tipos de argumentos: inteiros, ponto-flutuante, listas e dicionários

Polimorfismo

Habilidade de apresentar a mesma interface para formas diferentes, tipos diferentes de dados

- Por exemplo, métodos polimórficos podem aceitar tipos de dados diferentes e, dependendo dos mesmos, ter comportamentos diferentes

Em C++ para o polimorfismo, é necessária a definição de diversos métodos, um para cada configuração de tipos de argumentos

```

#include <iostream>
using namespace std;

void f(int x, int y) {
    cout << "values: " << x << ", " << y << endl;
}

void f(int x, double y ) {
    cout << "values: " << x << ", " << y << endl;
}

void f(double x, int y ) {
    cout << "values: " << x << ", " << y << endl;
}

void f(double x, double y ) {
    cout << "values: " << x << ", " << y << endl;
}

int main()
{
    f(42, 43);
    f(42, 43.7);
    f(42.3, 43);
    f(42.0, 43.9);
}

```

Sobrescrita, sobrecarga, substituição

Relacionadas ao polimorfismo, temos algumas variações são bem similares, veja:

- Sobrescrita:** Método com mesmo nome, quantidade de parâmetros (e/ou tipos de parâmetros)
- Sobrecarga:** Método com mesmo nome, mas quantidade e tipos de parâmetros diferentes
- Substituição:** Quando uma subclasse reescreve um método definido na superclasse

Sobrecarga

O mecanismo chamado de sobrecarga (*overloading*) é utilizado quando se deseja que dois métodos de uma mesma classe possam ter o mesmo nome, desde que suas listas de parâmetros sejam diferentes, constituindo assim uma assinatura diferente de cada método.

A sobrecarga não gera conflito pois o compilador é capaz de detectar qual método deve ser escolhido a partir da análise dos tipos de argumentos do método.

A sobrecarga é uma aplicação do polimorfismo na Orientação a Objetos.

```

>>> 4 + 5
9
>>> 3.8 + 9
12.8
>>> "Peter" + " " + "Pan"
'Peter Pan'
>>> [3,6,8] + [7,11,13]
[3, 6, 8, 7, 11, 13]

```

Em Python, o operador '+' é utilizado tanto para somar inteiros e ponto-flutuante e concatenar strings e listas

Métodos mágicos, em Python

Em Python, um inteiro, ponto-flutuante, string, lista, dicionários, etc. São tratados como objetos. Inclusive, essas 'classes base' podem ser herdadas, como veremos mais adiante.

Portanto, vários operadores podem ser sobreescritos para operar em todos esses tipos de dados e, inclusive, sobre objetos criados por um programador.

Isso é feito através dos métodos mágicos.

Veja a tabela a seguir, que lista os operadores, tanto aritméticos quanto lógicos, e seus respectivos métodos mágicos para sobreescrita.

Operador	Método mágico
+	object.__add__(self, other)
-	object.__sub__(self, other)
*	object.__mul__(self, other)
//	object.__floordiv__(self, other)
/	object.__truediv__(self, other)
%	object.__mod__(self, other)
**	object.__pow__(self, other[, modulo])
<<	object.__lshift__(self, other)
>>	object.__rshift__(self, other)
&	object.__and__(self, other)
^	object.__xor__(self, other)
	object.__or__(self, other)

Considere a classe Length, ao lado, e a sobreescrita do operador '+', pelo método __add__. Os métodos mágicos __str__ e __repr__ também foram utilizados

Assim, podemos instanciar um objeto da classe Length. Ao printar o objeto o método __str__ é invocado

```

x = Length(4)
print(x)
y = eval(repr(x))

z = Length(4.5, "yd") + Length(1)
print(repr(z))
print(z)

```

```

class Length:
    _metric = {"mm": 0.001, "cm": 0.01, "m": 1, "km": 1000,
               "in": 0.0254, "ft": 0.3048, "yd": 0.9144,
               "mi": 1609.344 }

    def __init__(self, value, unit = "m"):
        self.value = value
        self.unit = unit

    def Converse2Metres(self):
        return self.value * Length._metric[self.unit]

    def __add__(self, other):
        l = self.Converse2Metres() + other.Converse2Metres()
        return Length(l / Length._metric[self.unit], self.unit)

    def __str__(self):
        return str(self.Converse2Metres())

    def __repr__(self):
        return "Length(" + str(self.value) + ", '" + self.unit + "')"

```

Ao utilizar a função `repr`, o método `__repr__` é invocado e retorna uma String que, quando passada para a função `eval`, causa a instanciação de um novo objeto

```
x = Length(4)
print(x)
y = eval(repr(x))

z = Length(4.5, "yd") + Length(1)
print(repr(z))
print(z)
```

```
class Length:
    __metric = {"mm": 0.001, "cm": 0.01, "m": 1, "km": 1000,
               "in": 0.0254, "ft": 0.3048, "yd": 0.9144,
               "mi": 1609.344 }

    def __init__(self, value, unit = "m"):
        self.value = value
        self.unit = unit

    def Converse2Metres(self):
        return self.value * Length.__metric[self.unit]

    def __add__(self, other):
        l = self.Converse2Metres() + other.Converse2Metres()
        return Length(l / Length.__metric[self.unit], self.unit)

    def __str__(self):
        return str(self.Converse2Metres())

    def __repr__(self):
        return "Length(" + str(self.value) + ", '" + self.unit + "')"
```

Sobrecarga de operadores

Com a sobrecarga do operador '+', pelo método `__add__`, pode-se somar dois objetos da classe `Length`, como se soma dois número inteiros ou em ponto-flutuante

```
x = Length(4)
print(x)
y = eval(repr(x))

z = Length(4.5, "yd") + Length(1)
print(repr(z))
print(z)
```

```
class Length:
    __metric = {"mm": 0.001, "cm": 0.01, "m": 1, "km": 1000,
               "in": 0.0254, "ft": 0.3048, "yd": 0.9144,
               "mi": 1609.344 }

    def __init__(self, value, unit = "m"):
        self.value = value
        self.unit = unit

    def Converse2Metres(self):
        return self.value * Length.__metric[self.unit]

    def __add__(self, other):
        l = self.Converse2Metres() + other.Converse2Metres()
        return Length(l / Length.__metric[self.unit], self.unit)

    def __str__(self):
        return str(self.Converse2Metres())

    def __repr__(self):
        return "Length(" + str(self.value) + ", '" + self.unit + "')"
```

Atributos e métodos de um objeto

Considere a classe `Conta`, com um construtor e um método `set_saldo`, que lança uma exceção caso se atribua saldo negativo.

Instanciação de um objeto da classe `Conta`.

Acesso direto a um atributo privado da classe `Conta`. Nesse caso não se utilizará a função `set_saldo`. Como evitar isso?

```
class Conta:
    def __init__(self):
        self.__saldo = 0

    def set_saldo(self, saldo):
        if saldo < 0:
            raise Exception('Error')
        self.__saldo = saldo

c1 = Conta()
c1.__Conta__saldo = 100000000
c1.titular = 'Fulano Ciclano'
```

Atributos e métodos de um objeto

Considere a classe Conta, com um construtor e um método set_saldo, que lança uma exceção caso se atribua saldo negativo.

Instanciação de um objeto da classe Conta.

Agora, adiciona-se um novo atributo a um objeto já instanciado. É possível isso em Java ou C++?

```
class Conta:  
    def __init__(self):  
        self.__saldo = 0  
    def set_saldo(self, saldo):  
        if saldo < 0:  
            raise Exception('Error')  
        self.__saldo = saldo  
  
c1 = Conta()  
c1.__saldo = 100000000  
c1.titular = 'Fulano Ciclano'
```

Definindo e limitando atributos

O uso de __slots__ faz a restrição dos atributos do objeto, não permitindo a criação de novos atributos

Não é permitida a criação de novos atributos nos objetos da classe Conta

```
class Conta:  
    __slots__ = ['_Conta__saldo']  
    def __init__(self):  
        self.__saldo = 0  
    def set_saldo(self, saldo):  
        if saldo < 0:  
            raise Exception('Error')  
        self.__saldo = saldo
```

```
c1 = Conta()  
c1.new_saldo = 100000000 # Erro
```

O comando *dir()*

Retornará todos os atributos do objeto.
No exemplo, retornará:

```
['__Conta__saldo', '__class__',  
 '__delattr__', '__dict__',  
 '__dir__', '__doc__',  
 '__eq__', '__format__',  
 '__ge__', '__getattribute__',  
 '__gt__', '__hash__',  
 '__init__',  
 '__init_subclass__', '__le__',  
 '__lt__', '__module__',  
 '__ne__', '__new__',  
 '__reduce__', '__reduce_ex__',  
 '__repr__', '__setattr__',  
 '__sizeof__', '__str__',  
 '__subclasshook__',  
 '__weakref__', 'set_saldo']
```

```
class Conta:  
    def __init__(self):  
        self.__saldo = 0  
    def set_saldo(self, saldo):  
        self.__saldo = saldo
```

```
c1 = Conta()  
dir(c1)
```

Exercício:
Crie uma variável qualquer, x por exemplo, faça dir(x) e veja os atributos

O método mágico __call__

Permite que um objeto seja chamado diretamente, sem a necessidade de invocar um método específico

```
# a constant function  
p1 = Polynomial(42)  
  
# a straight Line  
p2 = Polynomial(0.75, 2)  
  
# a third degree Polynomial  
p3 = Polynomial(1, -0.5, 0.75, 2)  
  
for i in range(1, 10):  
    print(i, p1(i), p2(i), p3(i))
```

```
class Polynomial:  
    def __init__(self, *coefficients):  
        self.coefficients = coefficients[::-1]  
  
    def __call__(self, x):  
        res = 0  
        for index, coeff in enumerate(self.coefficients):  
            res += coeff * x** index  
        return res
```

os objetos da classe Polynomial são chamados diretamente.



Herança de classes base

As listas, em Python, têm o método `pop()`, mas não têm o método `push()`. Na realidade o método `push` é chamado de `append()`.

Caso se queira que uma lista tenha explicitamente um método `push`, pode-se herdar a classe base de listas (`list`) e adicionar esse método

```
y = [3,4]  
dir(y)  
x = Plist([3,4])  
x.push(47) ——————  
print(x)  
dir(x)
```

```
class Plist(list):  
    def __init__(self, l):  
        list.__init__(self, l)  
  
    def push(self, item):  
        self.append(item)
```

Agora, pode-se chamar o método `push()` para adicionar um elemento à lista.



Paradigma Orientado a Objetos : Python

Conceitos Iniciais:

Definição de classe:

```
class Pessoa:  
    def __init__(self, cpf, nome):  
        self.cpf = cpf  
        self.nome = nome
```

Objetos:

```
p1 = Pessoa('123.456.789-10', 'João da Silva')
```

Atributos são o `self.cpf` e `self.nome`

Atributos de classe nesse exemplo seria o `__total_pessoas`

```
class Pessoa:  
    __total_pessoas = 0  
    def __init__(self, cpf, nome):  
        self.cpf = cpf  
        self.nome = nome  
        Pessoa.__total_pessoas += 1  
    def get_total_pessoas(self):  
        return Pessoa.__total_pessoas  
  
p1 = Pessoa('123.456.789-10', 'Bissexto')  
print(p1.get_total_pessoas()) #erro  
print(Pessoa.get_total_pessoas(p1)) #OK
```

1

Métodos

```
from datetime import datetime  
class Pessoa:  
    def __init__(self, cpf, nome, data_nascimento):  
        d, m, a = data_nascimento.split("/")  
        self.cpf = cpf  
        self.nome = nome  
        self.data_nascimento = datetime(a, m, d)  
  
    def get_data_nascimento(self):  
        return self.data_nascimento.strftime("%x")
```

Construtores

```
class Medicamento:  
    def __init__(self, nome):  
        self.nome = nome
```

Destrutores

```
class A:  
    def __del__(self):  
        print("A has been destroyed")  
  
minhaClasse = A()  
del minhaClasse
```

A has been destroyed

Visibilidade de atributos

```
class A():
    def __init__(self):
        self.__priv = "I am private"
        self._prot = "I am protected"
        self.pub = "I am public"
```

Limitando os atributos

```
class Conta:
    __slots__ = ('_Conta__saldo')
    def __init__(self):
        self.__saldo = 0

c1 = Conta()
c1.new_saldo = 100000 #Erro
c1._Conta__saldo = 100000 #permitido
```

Decorators

```
class Conta:
    __slots__ = ('_Conta__saldo')
    def __init__(self):
        self.__saldo = 0
    @property
    def saldo(self):
        return 'R$ {:.2f}'.format(self.__saldo)
    @saldo.setter
    def saldo(self, novo_saldo):
        if novo_saldo > 0:
            self.__saldo = novo_saldo
        else:
            raise Exception()
    @classmethod
    def contas_instanciadas(cls):
        return '{} contas ativas'.format(cls.__contas)

c1 = Conta()
c1.saldo = 100 # chama setter
c1.saldo # chama getter
# R$ 100
c1.saldo = -100 #Exception
print(Conta.contas_instanciadas())
```

```
c2 = Conta()
print(c2.contas_instanciadas())
```

```
1 contas ativas
2 contas ativas
```

Herança em OO

```
class Pessoa:
    def __init__(self, nome):
        self.nome = nome

class Paciente(Pessoa):
    def __init__(self, nome, med_id):
        super().__init__(nome)
        self.med_id = med_id

class Medico(Pessoa):
    def __init__(self, nome, id_func):
        super().__init__(nome)
        self.id_func = id_func
```

Reo 4 - Paradigma Funcional

- Aulas
 - [Aula 1 - Paradigma Funcional : Introdução](#)
 - [Haskell](#)
- [Atividade Avaliativa](#)

Paradigma Funcional : Introdução

[video-aula](#)

Introdução

O paradigma funcional trata a computação como avaliação de funções matemáticas.

Esse estilo de programação é suportado por linguagens de programação funcional, ou linguagens aplicativas.

Linguagens funcionais possuem alto nível de abstração e estilo declarativo: especifica-se o que deve ser computado ao invés de como.

Alguns exemplos de linguagens funcionais: LISP, Scheme, ML e Haskell.2



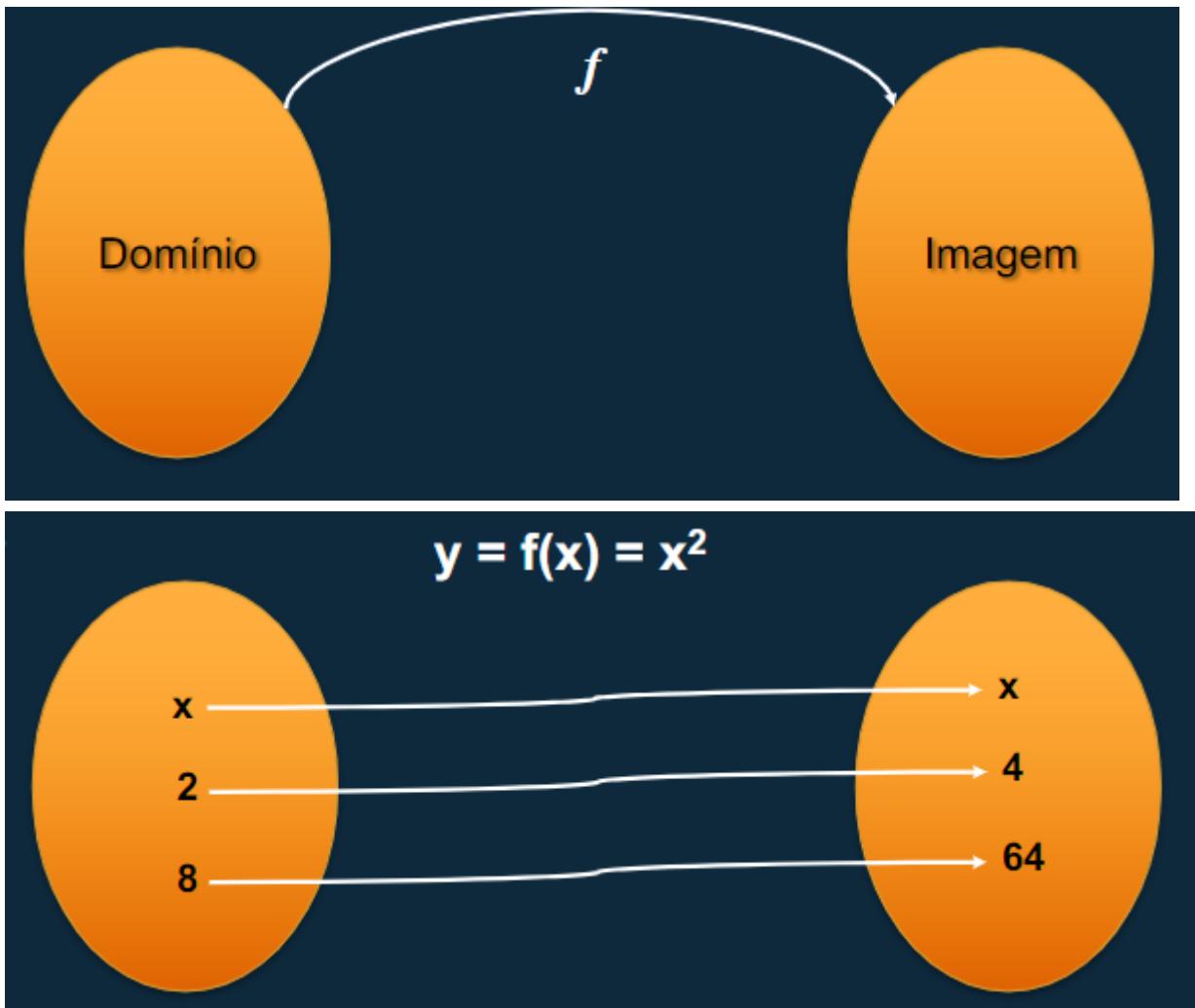
Funções matemáticas

Uma função matemática é um mapeamento de membros de um conjunto, chamado de conjunto **domínio**, para outro, chamado de conjunto **imagem**.

As funções são geralmente aplicadas a um elemento em particular do conjunto domínio, fornecido como um **parâmetro** para a função.

Uma função leva a, ou retorna, um elemento do conjunto imagem.

Em funções matemáticas, a ordem de avaliação de suas expressões de mapeamento é controlada por recursão e expressões condicionais, e não por sequência e repetição iterativa, como nas linguagens imperativas



Fundamentos da Programação Funcional - I

O objetivo do projeto de uma linguagem de programação funcional é **mimetizar funções matemáticas ao máximo possível**.

Em uma linguagem imperativa, uma expressão é avaliada e o resultado é armazenado em uma **posição de memória**, representada como uma variável em um programa.

Uma linguagem de **programação puramente funcional não usa variáveis**, nem sentenças de atribuição. Sem variáveis, as construções de iteração não são possíveis, já que elas são controladas por variáveis

Fundamentos da Programação Funcional - II

Na programação funcional, as repetições devem ser especificadas com recursão em vez de estruturas de repetição.

Uma linguagem funcional fornece:

- um conjunto de **funções primitivas**;
- um conjunto de **formas funcionais** para construir funções complexas a partir das funções primitivas;
- uma **operação de aplicação de função**;
- alguma estrutura ou **estruturas para representar dados**

Transparência Referencial

Programa funcional não tem 'estado'

Não tem atribuição: o programador não precisa se preocupar com variáveis

Dada uma função, podemos substituí-la por seu valor de retorno sem causar impacto na aplicação

O resultado de uma função é determinado unicamente por seus valores de entrada. Coisa alguma fora da função pode afetar a sua saída.

Isso é não tem efeito colateral!!

Uso da recursão

Principal causa da perda de performance, pois é computacionalmente caro realizar a recursão.

Se recursão for *de cauda* interpretador por mudar para iteração.

Pesquise o que significa recursão de cauda

Indicado é sempre tentar **recursão de cauda**

Funções Simples

```
cube(x) ≡ x * x * x, x ∈ ℝ  
cube: ℝ → ℝ
```

Funções Simples

- ◆ $\text{cube}(x) \equiv x * x * x, \quad x \in \mathbb{R}$
- ◆ $\text{cube}: \mathbb{R} \rightarrow \mathbb{R}$
- ◆ O símbolo \equiv significa é *definida como*

Exemplo de função que calcula o cubo de x. x pertence ao domínio dos números reais

O resultado da função cube depende apenas de x. Nenhum outro valor interfere no resultado: não tem efeito colateral

Funções Lambda

Alonzo Church, 1941, especificou funções não nomeadas

$\lambda(x)x*x*x$
 $(\lambda(x)x*x*x)(2)$

Resulta em 8

São funções que, geralmente, são utilizadas num escopo menor e, portanto, não precisam de um nome para ser referenciada em contexto mais amplo. λ é uma letra grega, nomeada lambda.

$\lambda(x)x * x * x$
 $(\lambda(x)x * x * x)(2)$
Resulta em 8

Formas Funcionais

Nem tudo se resolve com funções simples, como a função cubo. Então, as linguagens funcionais permitem as funções de ordem superior. Exemplos:

- Composição de funções
- Aplicar-a-todos

Composição de funções

$$h \equiv f \circ g$$

$$f(x) \equiv x+2$$

$$g(x) \equiv 3*x$$

$$h(x) \equiv f(g(x))$$

$$h(x) \equiv (3*x)+2$$

 $h \equiv f \circ g$

$f(x) \equiv x + 2$
 $g(x) \equiv 3 * x$

Na composição de funções, duas ou mais funções simples, são compostas para formar uma função mais complexa

$h(x) \equiv f(g(x))$
 $h(x) \equiv (3 * x) + 2$

Aplicar a todos

Denotada como α recebe uma única função como parâmetro e uma lista de argumentos

```
f(x) ≡ x*x  
α(f, (2,3,4))
```

Resulta em (4,9,16)

Atividade Avaliativa

Paradigma Funcional : Exemplos em Haskell