Programação Orientada À Objetos e Cálculo Lambda em Python

Paradigmas da Computação - Atividade 3

Felipe de Campos Santos - 17200441
Universidade Federal de Santa Catarina
10/02/2021

Antes de falarmos especificamente sobre Programação Orientada à Objetos em Python, é importante entendermos o que é POO (Programação Orientada à Objetos), suas aplicações e pontos positivos!

Programação Estruturada X Programação Orientada à Objetos

Anos atrás, o Programação Estruturada (PE) era a mais presente na vida dos programadores, principalmente com a popularização da Linguagem C. Basicamente, essa dita o passo-a-passo a ser seguido pelo programa, trazendo um ponto muito positivo em relação ao desempenho dos mesmos, principalmente quando se leva em consideração que o poder de processamento dos computadores da época eram bem baixos comparados aos atuais.

Com o aumento do poder de processamento e evolução das linguages de programação, a POO começou a ocupar o lugar onde antes a PE reinava, pois trouxe com ela diversas evoluções necessárias no contexto de aplicações modernas, que facilitam a vida dos programadores, como a reutilização de código e representação do sistema mais próximo do que vemos na vida real.

POO e seus pilares

Assim como comentado acima, a POO trouxe a possibilidade de representar um sistema mais próximo do que vemos na vida real. Isso trouxe impactos positivos em diversos setores da computação: otimização da geração de código, menos reescrita, aumento da curva de aprendizagem em geral, entre outras coisas.
Para ser considerada uma linguagem que segue o paradigma de POO, a linguagem precisa atender principalmente à 4 pilares, sendo eles:

- **Abstração:** Fala muito sobre se concentrar no pontos mais importantes e necessários, deixando que os detalhes mais genéricos aconteçam “por debaixo dos panos”. Os pontos mais importantes podem ser resumidos em:
    - *Identidade:* Diferencia os objetos entre eles, permitindo a criação de objetos diferentes dentro da mesma classe de objetos
    - *Propriedades:* Dita as características dos objetos em si
    - *Métodos:* Dita as ações que os objetos podem “executar”

- **Encapsulamento:** É uma camada de proteção aos objetos e suas propriedades, principalmente à aquelas que devem permanecer privadas ao objeto, além de ter uma participação importante na abstração. Dentro da POO são usados *getters* e *setters* (veremos mais adiante) para prover o encapsulamento, evitando com que as propriedades de um objeto sejam acessadas diretamente por um terceiro.

- **Herança:** Propriedade de POO muito importante na otimização da geração do código. Permite que seja evita a reescrita de código que seja comum a mais de uma parte do programa.

- **Polimorfismo:** Pilar ligado à Herança, permitindo que coisas herdadas dos ancestrais possam funcionar de forma diferente no objeto em questão.

Não se preocupe se por enquanto as coisas estão abstratas demais, nos exemplos ficará tudo mais claro!

Python e POO

Para que os exemplos fiquem mais lúdicos, vamos fingir que estamos criando um sistema para uma loja de eletrônicos, ok?

Comecemos criando uma *Classe* (pense como um grande grupo que ditará os objetos) e definindo suas *propriedades* (características dos objetos dessa classe). E pra isso, nossa loja comecará a vender computadores!
Agora, uma loja de computadores tem que mostrar pro cliente, no mínimo, a marca e o valor do computador, certo? Sim, na vida real seria muito mais! Processador, RAM, etc, mas deixaremos isso pra depois, ok?

Então vamos declarar nossa classe de computadores, isso é bem simples. Comece com a palavra *class*, que vai dizer ao código “olha só, aqui vou detalhar o que quero da minha classe!” e com o nome da classe depois, seguido de dois pontos.

```
class Computador:


    def __init__(self, modeloLoja, valorLoja):
        self.modelo: str = modeloLoja
        self.valor: int = valorLoja
```

Pronto! Nossa classe de computadores já tem duas propriedades (atributos): seu modelo e se valor. A função __init__(self): é responsável por sinalizar ao pyton que essa será o **construtor** dessa classe, ou seja, é a função responsável pela criação dos objetos. Essa função receberá dois parâmetros:

- modeloLoja: modelo que a loja está vendendo, representado por uma palavra (str = string = palavra)
- valorLoja: valor pelo qual a loja esta vendendo, representado por um numero decimal (float = numero com ponto flutuante)

e esses parametros serão associados aos atributos do objeto computador, seu modelo e seu valor.

Agora, vamos colocar dois computadores à venda na loja, shall we?
Os computadores serão os seguintes:

- Computador 1
    - Modelo: SUPER PC
    - Valor: R$1000,00
- Computador 2
    - Modelo: LÉPItop
    - Valor: R$1200,50

Nó codigo, isso é fácil! Fica assim:

```
computador1 = Computador('SUPER PC', 1000.00)
computador2 = Computador('LÉPItop', 1200.50 )
```

Pronto! Agora temos dois objetos da classe Computador instanciados! Pra acessar os atributos deles também é bem simples: você coloca o identificador do objeto (node dado pra ele) seguido de um ponto e do nome do atributo que voce quer. Ou seja, se rodarmos o código

```
print(computador1.modelo)
print(computador2.valor)
```

o que teremos no terminal de saida será

> SUPER PC
> 1200.5

Agora vamos falar um pouco sobre os métodos da classe, e pra isso vamos focar um pouco só no objeto Computador em si e sair um pouco da nossa loja, ok?
Qual a primeira coisa que a gente faz quando precisa usar um computador? A gente liga ele, certo? Então vamos criar um método da classe Computador que serve para ligar nosso Computador!

Sempre que formos criar uma função, a gente tem que definir ela, então usamos a palavra reservada *def*. Intuitivo, né?
E para que nossa função seja útil, vamos adicionar mais um atributo nos nossos objetos Computador que vai indicar o estado dele: ligado ou desligado.
Pra isso, vamos adicionar o atributo “ligado” que será booleano, ou seja, pode receber apenas dois valores: True e Falso, respectivamente significando que o computador esta ligado ou desligado. E como padrão, ele será criado desligado. Nossa classe Computador agora está assim:

```
class Computador:

    def __init__(self, modeloLoja, valorLoja):
        self.modelo: str = modeloLoja
        self.valor: float = valorLoja
        self.ligado: bool = False
```

Perceba que o construtor não recebe um parâmetro para ver se o computador está ligado ou não, isso porque padronizamos que ele sempre vai ser criado desligado!

Agora, embaixo do construtor vamos criar duas funções, uma pra ligar o computador e outra pra desligar o computador. Ficará assim

```
class Computador:

    def __init__(self, modeloLoja, valorLoja):
        self.modelo: str = modeloLoja
        self.valor: float = valorLoja
        self.ligado: bool = False

    def ligar(self):
        if(self.ligado):
            print('O computador ja está ligado!')
        else:
            self.ligado = True

    def desligar(self):
        if(not self.ligado):
            print('O computador ja está desligado!')
        else:
            self.ligado = False
```

Perceba que já colocamos também uma verificação de se é possível ou não ligar/desligar o computador, caso ele já esteja no estado pedido.

Agora, depois de criarmos nossos objetos de computador, podemos liga-los e desliga-los da seguinte maneira:

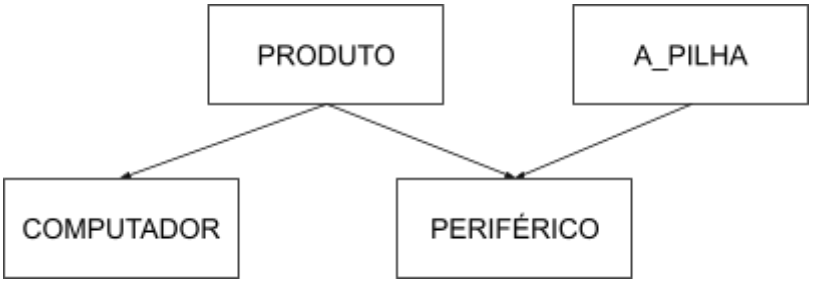
```
computador1 = Computador('SUPER PC', 1000.00)
computador1.ligar()
computador1.desligar()
```

Se, após ligar o computador um fizessemos uma chamada

```
print(computador1.ligado)
```

receberiamos um **TRUE** no terminal de saída.

Agora vamos melhorar um pouco nossa loja, porque ela era pra ser uma loja de eletrônicos mas por enquanto ela só vende computadores, precisamos de mais diversidade aqui.
Pra isso, vamos separar nossos produtos da seguinte maneira:



Ou seja, nossos computadores e nossos periféricos serão uma subclasse de Produto, e os periféricos também serão uma subclasse de A_Pilha.

Nossas classes agora são assim:

```
class Produto:

    def __init__(self, codigo, valor):
        self.codigo: int = codigo
        self.valor: float = valor

class A_Pilha:

    def __init__(self, tipoPilha, quantidadePilha):
        self.tipoPilha: str = tipoPilha
        self.quantidadePilha: int = quantidadePilha

class Periferico(Produto, A_Pilha):

    def __init__(self, tipo, codigo, valor, tipoPilha, quantidadePilha):
        super().__init__(codigo, valor)
        A_Pilha.__init__(self, tipoPilha, quantidadePilha)
        self.tipo: str = tipo

class Computador(Produto):

    def __init__(self, modelo, valor):
        self.modelo: str = modelo
```

```
        self.ligado: bool = False

    def ligar(self):
        if(self.ligado):
            print('O computador ja está ligado!')
        else:
            self.ligado = True

    def desligar(self):
        if(not self.ligado):
            print('O computador ja está desligado!')
        else:
            self.ligado = False
```

Agora vamos falar um pouco sobre Polimorfismo!
Primeiro, o que é isso?
Polimorfismo é quando temos duas funções com mesmo nome mas que realizam coisas diferentes, uma na classe pai (chamada de superclasse) e outra na classe filha (chamada de subclasse). Ou seja, uma funçao com mesmo nome mas mais de uma forma (dai vem o nome poli -> varias / morfos -> formas).

Pra exemplificar, vamos focar apenas nas classes Produto e Computador, e digamos que a gente deseja uma função que, quando chamada, nos retorne “Sou um produto” caso seja apenas um produto e “Sou um computador” caso o produto seja também um computador, e que essa função terá o nome “definir”.

```
class Produto:

    def __init__(self, codigo, valor):
        self.codigo: int = codigo
        self.valor: float = valor

    def definir(self):
        print('Eu sou um produto!')
class Computador(Produto):

    def __init__(self, modelo, valor):
        self.modelo: str = modelo
        self.ligado: bool = False

    def definir(self):
        print('Eu sou um computador!')
```

Ou seja, a classe Computador, por mais que seja uma subclasse de produto, tem seu próprio metodo “definir”. Sem o método “definir” na classe Computador, poderiamos instanciar um objeto c1 = Computador() e chamar c1.definir(), mas esse imprimiria “sou um produto!”

Composição e agregação

Composição e agregação são prioridades importantes na programação orientada à objetos por vários motivos, como melhorar a abstração e impedir a reescrita de código. Mas o que são e qual a diferença entre essas duas propriedades?
Bom, elas de maneira geral são uma relação de pertencimento, porém com “forças” diferentes. Pense dessa maneira:

- Em um banco, uma conta **só pode existir** com um cliente, ou seja, é impossível uma conta não ter um cliente que seja dono dela. Então a conta e o cliente tem uma relação de composição (a conta é composta pelo cliente)
- Ainda no banco, a conta tem um histórico de transações. Porém, mesmo sem ter um histórico, essa conta **ainda pode existir**. Então a conta e o histórico tem uma relação de agregação (o histórico faz parte da conta, mas a conta ainda existe sem ele).

No código, não existe muita diferença. Para exemplificar isso, faremos algumas mudanças nas classes A_Pilha e Periférico. Agora, elas ficarão assim:

```
class Pilha:

    def __init__(self, tipoPilha, quantidadePilha):
        self.tipoPilha: str = tipoPilha
        self.quantidadePilha: int = quantidadePilha

class Periferico(Produto):

    def __init__(self, tipo, codigo, valor, pilha):
        super().__init__(codigo, valor)
        self.tipo: str = tipo
        self.pilha: Pilha = pilha
```

Para fins de melhorar a sintaxe do programa, trocamos o nome da classe de “A_Pilha” para apenas “Pilha

Perceba agora que a classe periférico tem um atributo “pilha” que recebe um objeto pilha como valor. Essa é uma relação de agregação, pois o periférico pode existir sem uma pilha (caso ele seja alimentado via bateria do aparelho por exemplo)!

Para instanciar um objeto Periferico com uma pilha, faremos o seguinte:

```
pilha = Pilha('AAA', 4)
```

```
periferico = Periferico('mouse', 1, 100.00, pilha)
```

E se quisermos saber quantas pilhas vão nesse mouse, acessamos o atributo "quantidadePilha" do objeto Pilha desse Periferico, desse jeito:

```
periferico.pilha.quantidadePilha
```

Classes e metodos abstratos e estáticos

Mais nomes que fazem parecer que as coisas sejam complicadas, mas depois que tu entende fica fácil! Vem comigo:

Vamos tomar como exemplo nossa superclasse Produto e suas subclasses Computador e Periferico. Suponhamos que a gente tenha um array em nosso código que guarda todos os *produtos* que temos na nossa loja. Ao percorrermos esse array, vamos encontrar objetos apenas dos tipos Computador e Periferico, certo? Não faz sentido termos algum produto do tipo Produto, porque a gente criou essa classe apenas para que pudessemos usar a Herança para simplificar nosso código, mas nunca vamos de fato instanciar um objeto do tipo Produto. Porém, por mais que a gente nunca vá instanciar, ainda é possível que seja instanciado, certo? Então não faria sentido a gente fazer algo que tornasse impossível instanciar um objeto da classe produto? Assim evitariamos possíveis erros caso, por exemplo, a gente pare de trabalhar nessa loja e um outro programados passe a dar manutenção no nosso código. É para isso que servem classes abstratas!

Do jeito que nosso código está agora, se eu escrever as seguintes linhas:

```
produto = Produto(1, 100.49)
print(produto.valor)
```

e rodar o programa, teremos o seguinte na saída do terminal:
> 100.49

e queremos chegar aqui
> TypeError: Can't instantiate abstract class Produto with abstract method definir
mas ainda ser possível instanciar objetos das classes Computador e Periferico.

Pra isso, faremos uma importação de um pacote em python responsável por permitir isso. Na primeira linha do nosso código, colocaremos

```
from abc import ABC, abstractmethod
```

abc é *abstract base class* (classe base abstrata)

e nossa classe Produto ficará assim

```
class Produto(ABC):
    def __init__(self, codigo, valor):
        self.codigo: int = codigo
        self.valor: float = valor

    @abstractmethod
    def definir(self):
        print('Eu sou um produto!')
```

ou seja, colocaremos ABC como um dos parâmetros de construção da classe, e nos métodos dela colocaremos a anotação @abstractmethod, indicando que as subclasses dessa superclasse tenham que implementar essa função também!

Metodos estáticos

Vamos supor que queremos manter um controle de quantos computadores e perifericos temos na nossa loja. Como faremos isso? Bom, um jeito seria criar uma classe “Estoque”, instanciar um objeto dessa classe com atributos “n_computadores” e “n_perifericos” e sempre que criassemos um Computador e um Periférico a gente somasse uma unidade nesses atributos. Parece complicado, né? Existe uma maneira mais fácil: Metodos estáticos.

Metodos estáticos são métodos de uma classe que estão ligados **à classe**, e não à um objeto instanciado dela.

Façamos a seguinte mudança nas nossas classes Computador e Periferico

```
...
class Periferico(Produto):
    n_perifericos: int = 0
    ## resto do código

class Computador(Produto):
    n_computadores: int = 0
    ## resto do código
...
```

Agora temos uma variável que vai contar quantos computadores e periféricos temos na nossa loja. Para incrementar o número sempre que criarmos algum produto novo, vamos alterar nossos metodos __init__ com uma incrementação de uma unidade nessa variável.

```
class Periferico(Produto):

    n_perifericos: int = 0

    def __init__(self, tipo, codigo, valor, pilha):
        super().__init__(codigo, valor)
        self.tipo: str = tipo
        self.pilha: Pilha = pilha
        Periferico.n_perifericos += 1

class Computador(Produto):

    n_computadores: int = 0
```

```
    def __init__(self, modelo, codigo, valor):
        super().__init__(codigo, valor)
        self.modelo: str = modelo
        self.ligado: bool = False
        Computador.n_computadores += 1
```

pronto, agora sempre que criarmos um computador ou um periférico, essas variáveis serão acrescidas de uma unidade.

Agora se rodarmos as linhas

```
comp = Computador('te', 1, 100.00)
print(comp.n_computadores)
```

teremos
> 1
no terminal de saída.

Porém, é estranho usar um objeto computador para acessar quantos computadores temos na nossa loja, certo? Vamos tornar essa variável privada então!

- Aqui fica um comentário: em python, não existe uma sintaxe para variáveis privadas. O que existe é uma convenção de nomeação, ou seja, sempre que uma variável começar com "_", o programador que está lendo deve saber que aquela variável é pra ser acessada apenas com a classe pai, e não com um objeto por ela instanciada.

Vamos adicionar um "_" no começo do nome das nossas variáveis, e criar um método para acessa-lá

```
class Periferico(Produto):

    _n_perifericos: int = 0

    def get_total_perifericos(self):
        return Periferico._n_perifericos
...
class Computador(Produto):

    _n_computadores: int = 0

    def get_total_computadores(self):
        return Computador._n_computadores
...
```

Agora, se tentarmos chamar

```
Computador.get_total_computadores()
```

O terminal de saída apresentará o seguinte erro
> TypeError: get_total_computadores() missing 1 required positional argument: 'self'
Porque não passamos um objeto do tipo computador como parâmetro na função (aonde seria o self), e com isso voltamos na estranheza que é ser necessário ter um objeto da classe para acessar o número total de objetos da mesma. Pra resolver isso, finalmente vamos colocar a mão na massa com um método estático!
Pra isso, de novo, vamos usar uma anotação de python:

```
    @staticmethod
    def get_total_computadores():
        return Computador._n_computadores
```

usamos a anotação @staticmethod para indicar que o metodo que vem a seguir é um método estático, e com isso podemos retirar o “self” de seus parêmetros também!

Agora, se realizarmos a chamada

```
Computador.get_total_computadores()
```

nosso programa não terá nenhum problema, e nos retornará o número de objetos Computador criados até então.

Código completo:

```
from abc import ABC, abstractmethod

class Produto(ABC):
    def __init__(self, codigo, valor):
        self.codigo: int = codigo
        self.valor: float = valor

    @abstractmethod
    def definir(self):
        print('Eu sou um produto!')

class Pilha:

    def __init__(self, tipoPilha, quantidadePilha):
        self.tipoPilha: str = tipoPilha
        self.quantidadePilha: int = quantidadePilha

class Periferico(Produto):

    _n_perifericos: int = 0

    @staticmethod
    def get_total_perifericos():
        return Periferico._n_perifericos

    def __init__(self, tipo, codigo, valor, pilha):
        super().__init__(codigo, valor)
        self.tipo: str = tipo
        self.pilha: Pilha = pilha
        Periferico._n_perifericos += 1

    def definir(self):
        print('Eu sou um periferico!')



class Computador(Produto):

    _n_computadores: int = 0
```

```
    @staticmethod
    def get_total_computadores():
        return Computador._n_computadores

    def __init__(self, modelo, codigo, valor):
        super().__init__(codigo, valor)
        self.modelo: str = modelo
        self.ligado: bool = False
        Computador._n_computadores += 1

    def definir(self):
        print('Eu sou um computador!')

    def ligar(self):
        if(self.ligado):
            print('O computador ja está ligado!')
        else:
            self.ligado = True

    def desligar(self):
        if(not self.ligado):
            print('O computador ja está desligado!')
        else:
            self.ligado = False


##  aqui embaixo colocar as chamadas que você quiser, como criação
#   de objetos, chamadas de método e etc.
##
```

Cálculo Lambda em Python

1. Quem foi o idealizador do cálculo lambda e quando ele foi proposto (aproximadamente)?

Alonzo Church, aproximadamente na década de 1930, como parte de um trabalho seu em fundamentos matematicos

2. Qual a relação entre cálculo lambda e máquina de Turing?

“calculo lambda esta para software assim como maquinas de turing estão para hardware”
Alan Turing provou em 1937 a equivalência entre uma Máquina de Turing e o Cálculo Lambda em termos de computabilidade.

3. O que é o cálculo lambda? Em que ele foi útil na computação?

Um conjunto de várias modelos baseados em notação pra função com o intuito de simplificar operadores de funções ou como funções podem ser combinadas para virar operadores.
Uma de suas importâncias é a maior facilidade em linkar a linguagem de programação de alto nível (funcional) e suas implementáveis em baixo nível, também mantém uma semântica simples e boa estruturação sintática

4. O que são variáveis livres (independentes)? E variáveis vinculadas (dependentes)? Cite exemplos de variáveis livres e vinculadas em uma expressão lambda.

Variáveis lívres são aquelas que não estão ligadas na função por uma abstração lambda. É possível fazer uma associação com variáveis de funções polinomiais para esse caso. Por exemplo, na função
λx.x, não temos variáveis livres, pois x está vinculado à λx. Já em λx.t, as variáveis de t (menos x caso haja) são todas variáveis livres. Ex, λx.(x+y) tem a variável livre y e a variável ligada x.
Em λx.ts são livres as variáveis resultantes da união de variáveis livres em t e em s.
Expressões lambda sem variáveis livres são ditas “fechadas” ou “combinadores”

5. O que significa currying em cálculo lambda? Exemplifique.

Qualquer função que recebe duas entradas, como por exemplo
(x, y) = 2*x*y ou (x, y) -> 2*x*y
pode ser reescrita em uma função equivalente que recebe uma unica entrada e retornar *uma outra função* que por sua vez tambem recebe uma entrada, ou seja, uma função de duas entradas vira uma fila de duas funções de uma entrada. Por exemplo
x -> (y -> 2*x*y)

Sem currying:
((x,y) -> 2 * x * y) (5,7) =
= 2 * 5 * 7 = 70

Com currying:
(x -> (y -> 2 * x * y)(5))(7)=
= (y -> 2 * 5 * y)(7) =
= 2 * 5 * 7 = 30

Com ou sem currying o resultado é o mesmo, mas com currying podemos “transformar uma variável em constante” (como foi feito com x no segundo passo, onde virou a constante 5)

6. O que significa uma expressão ser um "combinador", em cálculo lambda? Cite um exemplo de expressão que é um combinador.

Combinador é o nome dado para expressões lambda sem variáveis livres (onde todas as variáveis são ligadas ao operador de abstração λ.
Exemplo: λx.x (a função identidade) é uma expressão lambda fechada.

7. O que é aplicação e o que é abstração numa expressão lambda? Exemplifique.

A definição de abstração e aplicação vem das regras de definição de expressões lambda:

Seja Λ o conjunto das expressões lambda:

1. se x é uma variável, então x ∈ Λ
2. se x é uma variável e M ∈ Λ, então (λx.M) ∈ Λ *(abstração)*
3. se M, N ∈ Λ, então (M N) ∈ Λ *(aplicação)*

Abstração vem de juntar uma função à uma variável, ou seja, receber uma entrada e substitui-la na função. Por exemplo, a função f(x) = x + 2 pode ser abstraida em λx.(x+2) (λx.t com t sendo (x+2)), onde λx define a entrada e (x+2) a função a ser computada e retornada.

Já a aplicação pode ser derivada diretamente da regra de definição 3, onde uma aplicação ts representa uma função t com s com entrada, ou seja, t(s)

8. O que significa dizer que duas expressões lambda são α-equivalentes? Exemplifique.

Respondido no ponto 9 abaixo.

9. O que é a operação de α-conversão (α-renomeação)? Exemplifique.

É a alteração do nome de variáveis ligadas, que a princípi pode parecer uma simples substituição de uma letra mas sem o devido cuidado pode mudar completamente a função.
Exemplos:
A α-conversão de λx.x com y como substituto resulta em λy.y. Esses dois termos, um proveniente da α-conversão de outro, são termos α-equivalentes, ou seja, termos literalmente idênticos.
Os cuidado devem ser tomados em situações como λx.λx.x, que usando y como substituto, pode resultar em λy.λx.x mas *nunca* em λy.λx.y, que é um termo diferente do original.
Também deve ser cuidado com colocar a variável substituida no escopo de uma abstração diferente, por exemplo, se substituirmos o x por y em λx.λy.x, teriamos λy.λy.y, que é um termo diferente do original.

10. O que é Redução-Beta (β-Redução)?

Para entender β-Redução, primeiro é importante entender o conceito de substituição de

variáveis.
-> A notação t[x:=r] indica a substituição de x por r em t
A β-Redução, simplificadamente, pode ser resumida a “aplicar funções aos seus argumentos”. Do ponto de vista da computação pode ser entendida como um passo de computação, servindo como “facilitador” de uma função lambda. Tomemos como exemplo a função lambda abaixo
(λx.t)s
Sabemos que essa função se dará pela aplicação de s no termo x de t. Então, temos uma substituição equivalente dada por
t[x:=s]
ou seja, a notação (λx.t)s -> t[x:=s] é usada para dizer que (λx.t)s é beta-reduzido para t[x:=s].
Com isso, podemos demonstrar facilmente funções identidade (f(x) = x) e funções constante (f(x) = y), usando de:

- Para todo s, (λx.x)s -> x[x:=s] = s, ou seja, aplicado s retorna s, logo essa função é de fato a identidade.
- Para todo s, (λx.y)s -> y[x:=s] = y (visto que em y não existem x, com x!=y) então essa é uma função constante, independendo de s sempre retornará y.

11. O que significa dizer que uma expressão lambda está em sua "forma normal"? Exemplifique com uma expressão lambda em sua forma normal.

Dizer que uma expressão λ esta em sua forma normal é dizer que ela está em sua forma mais “simplificada” possível, ou seja, não existem mais β-reduções possíveis de serem feitas, ou seja, não ocorre mais nenhuma expressão do tipo (λx.t)M

12. Pesquise sobre o Combinador Y. O que é e o que ele faz? Descreva um pouco seu funcionamento.

O combinador Y é uma implementação de combinador de ponto fixo para cálculo lambda,
mas o que é um combinador de ponto fixo?
É uma função que satisfaz a equação y f = f(y f) pra todo f, ou seja, y quando aplicado à uma função f produz o mesmo resultado que f aplicado para o resultado da aplicação f para y, e é chamado assim porque, por definição x = y f é uma solução para a equação de ponto fixo x = f x, sendo que um ponto fixo de uma função f é um valor que não é alterado sob a aplicação da função f.

Sua importancia para a computação é que a implementação Y
Y = λf.(λx.f(x x))(λx.f(x x))
pode ser utilizada na implementação do paradoxo de Curry, que diz que calculo-λ não pode ser usado como sistema dedutivo pois permitir que uma expressão anonima represente zero é inconsistente na lógica.
Ao aplicar Y em uma exmpressão com uma variável, geralmentese tornara uma coisa sem fim, mas as coisas se tornam mais interessantes quando se uas Y em funções com duas ou mais variáveis, pois possibilita que a segunda variável seja usada como contador, dando a possibilidade da criação de coisas como loops em linguagens imperativas (while, do for). Desta forma, Y permite recursão simples, passando a própria função como segundo parametro dela mesma, sendo o primeiro parâmetro o contador ou delimitador da computação.