Funções

Agora vamos falar de algo que particularmente pra mim, eu gosto e uso bastante, por usar um paradigma de programação, onde esses carinhas reinam, funções que qualquer pessoa que utilize uma linguagem de programação conhece e ja trabalhou com elas o objetivo aqui e passar algumas praticas.

Aqui vale um atendo se você é do tipo de pessoa que conhece Paradigma de Programação Orientado a Objeto (POO), **e acredita que função e métodos são a mesma coisa, é tenho que dizer pra você: NÃO É NÃO, métodos são algo só de POO,** função é um pouco mais global em outros paradigmas.

Tamanho

Pequenas

Uma das primeiras coisas que eu lembro quando eu comecei com programação la em 2014, isso com HTML, CSS e JS, que um professor me frisou bem, que quando você for escrever tome cuida com o amanhã da linhas,, para que elas não tenham mais de 80 caracteres, o engraçado que nesse época nos so usávamos so o bloco de notas ele ainda, não tinha nos apresentado o Notepad++, então foi um pouco difícil imagine eu contando.

O importante é linhas com até 80 caracteres, por que ninguém quer ficar, que ficar rodando a barra horizontal do editor de texto, para ver uma parte do Código, ai volta pra ver o resto, lê mais um pouco, tem que rodar a barra lateral de novo, ai volta a barra lateral, e continua sim. Então eu lhe pergunto PORQUÊ, tem necessidade, não (esse tamanho foi definido por conta de Telas VT100, que tinha 24 linhas e 8, colunas)

Agora referente ao tamanho delas na vertical, Uncle Bob, ele aborta que de 2 a 4 linhas, isso pra não concordo completamente, **pra mim uma função com 10 linhas, dependendo da estrutura usada pode tão limpa quando uma 3 linhas.**

Bloco e Endentação

Primeiramente nos temos estruturas de controle e condicionais, é quem são elas **If, else, while e switch também se a linguagem tiver,** é aqui nos temos um direcionamento que faz todo sentido, essas devem dentro das suas condicionais **deveriam ter so uma uma linha de preferencia a chamada de uma função.**

Exemplo uma função que retorna um numero ao quadrado se for par, e se for impar ele retorna o numero ao cubo:

```
from typing import Union

number_four = 4
number_five = 5

def ruler_number_cube_square(number: Union[int, float]):
  try:
    if number % 2  is 0:
      return number**2
    else:
      return number**3
  except TypeError:
    raise TypeError(
      (
        "argument is not a number of type int or float, " +
        f"type from argument {type(number)}"
      )
    )


ruler_number_cube_square(number_four)
ruler_number_cube_square(number_five)
```

Vamos melhorar isso daqui

- Primeiro tenho duas Estrutura condicionais if-else e try-except, vou precisar separar elas: para que eu consiga retornar elas em so uma linha

```
number_four = 4
number_five = 5


def is_odd_even_response(number):
    return number**2 if number % 2 == 0 else number**3


def ruler_number_cube_square(number):
    try:
        return is_odd_even_response(number)
    except TypeError:
        raise TypeError(
            (
                "Argument is not a number of type int or float, " +
                f"type from argument is {type(number)}"
            )
        )


ruler_number_cube_square(number_four)
ruler_number_cube_square(number_five)
```

dentro da função eu acabo utilizando um operador ternário por or deter uma condição, o que ajuda a minha função ficar bem pequena

eu poderiam melhorar mais isso, separando as funções de calculo:

```
from typing import Union

number_four = 4
number_five = 5


def number_square(number: Union[int, float, complex]):
```

```
    return number**2


def number_cube(number: Union[int, float, complex]):
    return number**3


def is_odd_even_response(number):
    return number_square(number) if number % 2 == 0 else number_cube(number)


def ruler_number_cube_square(number: Union[int, float]):
    try:
        return is_odd_even_response(number)
    except TypeError:
        raise TypeError(
            (
                "Argument is not a number of type int or float, " +
                f"type from argument is {type(number)}"
            )
        )


ruler_number_cube_square(number_four)
ruler_number_cube_square(number_five)
```

E poderíamos melhorar muito mais usando elementos de programação funcional:

```
from typing import Callable, Any, Union

number_four = 4
number_five = 5

RealNumber = Union[int, float]
Number = Union[int, float, complex]


def number_square(number: Number) -> Number:
    return number**2


def number_cube(number: Number) -> Number:
    return number**3


def is_odd_even_response(
    number: RealNumber,
```

```
    res_even: Callable[[Number], Any],
    res_odd: Callable[[Number], Any]
) -> Any:
    return res_even(number) if (number % 2) == 0 else res_odd(number)


def ruler_number_cube_square(number: RealNumber) -> RealNumber:
    try:
        return is_odd_even_response(number, number_square, number_cube)
    except TypeError:
        raise TypeError(
            (
                "Argument is not a number of type int or float, " +
                f"type from argument is {type(number)}"
            )
        )


ruler_number_cube_square(number_four)
ruler_number_cube_square(number_five)
```

Coisas que nos temos que evitar, estruturas condicionais e de controles aninhadas em níveis, no livro, são recomendados até dois níveis, devemos evitar em todo o custo utilizar estruturas Hadouken como abaixo:

```
if condicao1:
    # Nível 1
    if condicao2:
        # Nível 2
        if condicao3:
            # Nível 3
            if condicao4:
                # Nível 4
                if condicao5:
                    # Nível 5
                    # Código a ser executado se todas as condições forem verdadeiras
                else:
                    # Nível 5
                    # Código a ser executado se a condicao5 for falsa
            else:
                # Nível 4
                # Código a ser executado se a condicao4 for falsa
        else:
            # Nível 3
            # Código a ser executado se a condicao3 for falsa
    else:
        # Nível 2
        # Código a ser executado se a condicao2 for falsa
```

```
else:
    # Nível 1
    # Código a ser executado se a condicao1 for falsa
```

Principio da Minima Responsabilidade - Talvez a coisa mais importante

Uma das coisas mais importante é que as nossas funções façam apenas uma coisa, quando elas param de fazer, mais de uma coisa, significa que temos espaço para quebra-la em duas ou mais funções.

```
from typing import Union

number_four = 4
number_five = 5

def ruler_number_cube_square(number: Union[int, float]):
  try:
    if number % 2  is 0:
      return number**2
    else:
      return number**3
  except TypeError:
    raise TypeError(
      (
        "argument is not a number of type int or float, " +
        f"type from argument {type(number)}"
      )
    )


ruler_number_cube_square(number_four)
ruler_number_cube_square(number_five)
```

A nossa função tinha N responsabilidades:

- Garantir que o Código roda ou retornar um erro personalizado
- Verificar se o numero e impar ou par
- E depender da resposta de cima executar o calcula especifico para cada caso

A para tudo essas responsabilidades nos abstraímos e criamos a função abaixo, veja com cada uma das responsabilidades esta distribuída para uma função (fica mais fácil ler as reponsabilidades de cima pra baixo, que encaixa certinho nas ordem das funções).

```
from typing import Union

number_four = 4
number_five = 5


def number_square(number: Union[int, float, complex]):
    return number**2


def number_cube(number: Union[int, float, complex]):
    return number**3


def is_odd_even_response(number):
    return number_square(number) if number % 2 == 0 else number_cube(number)


def ruler_number_cube_square(number: Union[int, float]):
    try:
        return is_odd_even_response(number)
    except TypeError:
        raise TypeError(
            (
                "Argument is not a number of type int or float, " +
                f"type from argument is {type(number)}"
            )
        )


ruler_number_cube_square(number_four)
ruler_number_cube_square(number_five)
```

Try e except

Vale avisa que **Try-expect ou try-catch**, dependendo da linguagem, onde você esteja, temos que entender que essas estruturas **equivalem a uma reponsabilidade, então deve ter funções especificas para testar tais condições.**

Nome

O **nome é um bom descritivo de qual é a responsabilidade de uma função e o que ela deveria se preocupar em realizar,** a partir de nome você conseguira direcionar com realizar a quebra da implementação. Por isso é importante investir tempo nisso e lembrar toda vez que você chegar em **“minha função faz isso e isso”, o “e” ja é um ótimo indicativo que você esta acumulando responsabilidades.**

Switch ou elif

Em muitos casos o utilização de Switch pode, apresenta um acumulo de responsabilidade, para a função pode isso buscamos evita-lo, se possível.

Vamos ao seguinte exemplo:

```
def money_pay(employee):

    if employee[type] == 'COMMISSONED':
        return calc_commissoned(employee)
    elif employee[type] == 'HOURLY':
        return calc_hourly(employee)
    elif employee[type] == 'SALARIED':
        return calc_salaried(employee)
    else:
        raise TypeError('Value not excepted')
```

o Principal motivo disso não parecer com e por que se algo mudar dentro de employee, passar para o proprio objeto a capacidade de se gerenciar, e manter o principio de aberto e fechado onde eu posso adicionar mais um tipo e não preciso mexer na regra do objetivo. e se o objeto for motificado eu não precise realizar alteração em todos as funções de calculo

```
from collections import namedtuple


def create_type(name, calculate):
    GenericType = namedtuple(
        'GenericType', ['name', 'calculate']
    )
    return GenericType(name, calculate)


def type_hourly_pay():

    def calculate(value):
        pass

    return create_type('hourly', calculate)


def type_salarie_pay(value):

    def calculate(value):
        pass

    return create_type('salarie', calculate)


def type_commissed_pay(value):

    def calculate(value):
        pass

    return create_type('commissed', calculate)


def employee(name, type, week_hours, salarie, job):
    Employee = namedtuple(
```

```
        'Employee',
        [
            'name',
            'type',
            'week_hours',
            'base_salarie',
            'job_function',
            'calculate'
        ]
    )

    return Employee(name, type.name, week_hours, salarie, job, type.calculate)


def create_type_employee(type):

    type_employee = {
        "COMMISSIONED": type_commissed_pay(),
        "SALARIED": type_salarie_pay(),
        "HOURLY": type_hourly_pay()
    }

    if type not in type_employee:
        raise ValueError('Invalid employee type')

    return type_employee[type]


def FactoryEmployee(name, type, week_hours, salarie, job):
    return employee(name, create_type_employee(type), week_hours, salarie, job)

def calcule_pay(employee):
    return employee.calculate(employee.salarie)
```

O que nos temos:

1. A função `create_type` é uma função auxiliar que cria um tipo genérico com base no nome e na função de cálculo fornecidos. Isso ajuda a evitar repetições de código.
2. As funções `type_hourly_pay`, `type_salarie_pay` e `type_commissed_pay` retornam instâncias dos tipos específicos com a função de cálculo correspondente. Essas funções são concisas e podem ser facilmente estendidas para adicionar lógica de cálculo real.

3. A função `employee` cria uma instância de `Employee` utilizando os valores fornecidos, incluindo o tipo de funcionário, o nome, as horas semanais, o salário base, o trabalho e a função de cálculo.
4. A função `create_type_employee` é responsável por retornar a instância correta do tipo de funcionário com base no tipo fornecido. Se um tipo inválido for fornecido, uma exceção `ValueError` será lançada.
5. A função `FactoryEmployee` é responsável por criar uma instância completa de um funcionário, utilizando as funções auxiliares mencionadas acima para criar o tipo de funcionário correto e, em seguida, chamar a função `employee` para criar o objeto `Employee`.

Agora se eu precisar adicionar uma tipo ele criou um tipo e adiciono na função create_type_employee, e é o método de calculo e inserido a partir do dentro da objeto.

Como uma narrativa

Agora vamos Falar sobre um principio importante que e a Leitura a partir da Regra Decrescente, onde temos, com forme nos vamos descendo no código deva ficar menos complicado, as principals partes devem ficar mais perto do topo. **Assim com uma lista de TO-DO, e preciso disso pra depois pode fazer isso**

por exemplo vamos pegar uma função de que realiza uma tratamento de dados:

```
from re import match
from pandas import DataFrame
from typing import Union
from forex_python.converter import CurrencyRates


def treat_remove_space(
    dataframe: DataFrame,
    column: str
) -> DataFrame:
```

```
    dataframe[column] = dataframe[column].apply(
        lambda cell: cell.strip()
    )

    return dataframe


def extract_name_coin(
    dataframe: DataFrame,
    column: str,
    name_new_column: str,
    regexp: str = '^[A-Z]{3}'
) -> DataFrame:
    dataframe[name_new_column] = dataframe[column].apply(
        lambda cell: match(cell, regexp)
    )
    return dataframe


def treatment_dataframe(dataframe: DataFrame) -> DataFrame:

    dataframe = treat_remove_space(dataframe, 'descriptorTransaction')
    dataframe = extract_name_coin(
        dataframe,
        'descriptorTransaction',
        'coin'
    )
    return dataframe


def convert_coin_to_real(
    dataframe: DataFrame,
    column_value: str,
    column_coin: str,
    name_column: str
):

    def convertor():
        c = CurrencyRates()
        return c.get_rate('USD', 'BRL')

    def calculate_value_real(value: Union[int, float]) -> float:
        return convertor() * value

    def check_is_coin_real(
        value: Union[int, float],
        coin: str
    ) -> Union[int, float]:
        return value if coin == 'BRL' else calculate_value_real(value)

    def convert(
        dataframe: DataFrame,
        column_coin: str,
```

```
        column_value: str,
        name_new_column: str
    ) -> DataFrame:
        dataframe[name_new_column] = dataframe.apply(
            lambda row: check_is_coin_real(
                row[column_value],
                row[column_coin]
            ),
            axis=1
        )

        return dataframe

    return convert(dataframe, column_coin, column_value, name_column)


def implementing_convert_coin(dataframe: DataFrame):
    return convert_coin_to_real(
        dataframe,
        'valueTransaction',
        'coin',
        "valueReal"
    )


def process_data(dataframe: DataFrame):
    dataframe = treatment_dataframe(dataframe)
    dataframe = implementing_convert_coin(dataframe)

return dataframe
```

Se você perceber lendo o Código chegamos o entendimento dos passos necessaries conseguimos construir a logica como se as peças fossem se encaixando pouco a pouco. para o computador na execução do programa a logica é ao inverso ele lê ela de baixo para cima

Parametrons de Funções

Se formos seguir o que ´é dito dentro do Clean Code, nos temos 5 formas de assinaturas de funções derivadas pela quantidade de parâmetros sendo:

- Nulas - Sem parâmetros
- Monades - Com uma parâmetro
- Díade - Com dois parâmetros
- Tríade - com três parâmetros
- Poliade - com 4 ou mais paramotors

Para o Uncle Bob, funções deveriam, **não ter parâmetros, e isso me deixa um pouco desconfortável ,** pois por exemplo.

O que é um função sem parâmetros?

Simples funções que alteram o estado de algo, ou seja geram efeitos colaterais, pois não sabemos ao certo o que ela esta realizando ou qual é o efeito disso, mexer com estado, pois eu não estou fazendo a manipulação de algo de forma direta.

E em segundo, isso faz muito sentido para linguagens multi-paradgimas ou de POO, como o Java onde houveram os exemplo, mas não para uma linguagem funcional onde as funções reinam, parâmetros acabam sendo uma coisa essencial.

A forma de resolver (pelo menos pelo livro)

A forma sugerida de resolver esse problema e simples ao invés de passar um monte de parâmetros, base um objeto que em capsula todos os parâmetros em um e base esse objeto ou algum tipo dado do tipo primitivo que tenha essa função.

Complexidade do Código

Outra coisa comentada **no livro é que a quantidade de parâmetros é igual a complexidade do código, geralmente,** e claro que para funções nos podemos

resolver isso em linguagens que permitem a utilização de valores padrões, definir uma valor padrão para aquela função é isso eventualmente ajuda a diminuir a complexidade da função.

Valores Boolean

Em alguns casos tem uma **variável booleana, com parâmetro, pode significar que sua função esta realizando mais de uma coisa e pode ser um indicativo que seja necessário abstrair.**

valores nulos

Parametrons com valores nulos como padrão, ou que esperam valores, nulos são um ótimo indicativo que algo esta errado, no caso de considerar v**alores nulos como padrão ou na assinatura significa que seja necessaire abstrair e entender o que realmente deveria ser esperado pela função, se não a um outro valor que possa substituir o nulo, ou até mesmo se o conceito daquele parâmetro deveria ser revisado.**

Valores nulos com Saida

Dependendo da função ter **valores nulos pode representar uma problema isso quando esse valor e esperado no seu Código, pois valores nulos, pode gerar exceções**, sendo necessários que outras funções tratem no caso do valor retornar como nulo.

Se for um reposta de erro sempre prive, por retornar exceções do que valores que se passem por outras coisas

Evite Efeitos Colaterais

Efeitos colaterais podem ser causados por ações como modificar variáveis globais, alterar arquivos, fazer chamadas de rede, interagir com bancos de dados, exibir informações na tela, manupipulação de estado entre outros. Eles podem ser intencionais ou acidentais, mas, em geral, é recomendado minimizar os efeitos colaterais em código limpo e de boa qualidade.

A abordagem preferencial é escrever funções puras, que não possuem efeitos colaterais e produzem apenas resultados com base em seus argumentos de entrada. Essas funções não modificam o estado do programa ou de variáveis globais. Torna-las mais legiveis e facies de testar também pode ser uma forma de minimizar esses efeitos quando não for possivel evitar.

Consulta e Saida

No livro é abortado, ou o seu Código, ou a sua função de de consulta ou e de Saida.

Saida: Manipula, algo e retorna uma valor novo objeto daquela instancia, ou realizam um calculo ou operação.

Consulta: Altera o estado de algo e tem função de comunicação, que tem o objetivo de gerar logs 9ou coisa do tipo, e nelas não faz sentido eu ter um retorno, principalmente como booleano e etc.

```
def delete_paga(render, page):
    render.pages(page).delete()
    return True
```

Por que mesmo que nos tenhamos o resultado não sabemos se realmente a pagina foi deletada ou não, a melhor forma é não ter uma retorno para essa função. O que padrão no python é igual a None e em Java seria Void

```
def delete_page_to(render, page) -> None:
    render.pages(page).delete()
```

Evite Repetição

Essa e bem simples, evite repetições no seu código sempre que puder, se houver repetição analise se não existe a oportunidade de abstrair o que esta repetido para uma função. Lembre que estamos avaliando a logica e não os valores das variáveis, pois as mesmas podem ser substituídas por parâmetros na assinatura da função.