IFC – BLUMENAU
Gabriel Eduardo Lima

UMA VISÃO INTRODUTÓRIA DE REFLEXÃO EM PYTHON

2021

OBJETIVO

O presente trabalho tem como objetivo introduzir alguns conceitos e recursos reflexivos da linguagem de programação Python. Inicialmente são abordados alguns tópicos teóricos, sendo que esses servem para induzir o leitor aos exemplos de código formulados e apresentados na seção final do trabalho. Pretende-se que esse trabalho seja apresentado como meio avaliativo na disciplina de Programação Orientada a Objetos II do curso de Bacharelado em Ciências da Computação ofertado pelo IFC – Campus Blumenau.

Como ressalto é necessário observar que o trabalho atual não pretende segue padrões de formatação a rigor, uma vez que o texto solicitado na disciplina visa ser mais simples. Todavia para permitir que os mais diversos leitores consigam entender os aspectos do trabalho, optou-se por escrever alguns tópicos de maneira mais detalhada, garantindo uma visão inicial da reflexão de software.

DESENVOLVIMENTO

1. Sobre a reflexão

De maneira resumida a reflexão no contexto da computação refere-se a capacidade de um sistema em ter conhecimento de si próprio e do ambiente no qual está inserido. Além disso faz-se necessário que esse sistema tenha habilidade de manipular esse ambiente e estruturas do programa que faz parte.

A descrição em um primeiro momento pode causar confusões para aqueles que não estão habituados ao assunto, todavia trazendo em palavras simples pode-se dizer que a reflexão nada mais é do que “realizar programação sobre a própria programação”, aspecto muitas vezes também referido como metaprogramação.

Sendo assim, sistemas reflexivos contam com todo uma estruturação que vai além da programação usual, geralmente efetuada para resolver algum problema de um domínio. Essa adição refere-se a elementos capazes de compreender e manipular os próprios aspectos do programa, o que não necessariamente interage diretamente com o domínio no qual está inserido. Visando tornar o entendimento sobre reflexão mais conciso, considere o exemplo hipotético descrito logo abaixo, e em seguida tente compreender a discussão realizado sobre esse contexto.

> “Você está desenvolvendo um sistema simples, mas que contém diversos objetos de classes diferentes. Ao fim desse processo você deseja mapear rapidamente suas classes para um banco de dados visando implementar persistência em seu sistema e salvar os dados de seus objetos. Nesse caso você simplesmente precisa saber o nome da classe e seus atributos (nesse exemplo não vamos considerar os tipos dos dados, objetivando um exemplo simples) para criar as tabelas e permitir que os dados dos objetos sejam armazenados corretamente.“

Nesse exemplo simples, você poderia facilmente criar um programa para criar as tabelas no seu banco de dados, onde você simplesmente copia o nome da classe e seus atributos para a lógica de seu programa de persistência, e em seguida armazena os dados de seus objetos no Banco.

Todavia considere um caso em que você têm um sistema com mais de 50 classes onde cada uma por sua vez possui em média 8 atributos. Observe o imenso trabalho que seria necessário para desenvolver uma tarefa simples e essencial ao seu sistema. E se você criasse um novo sistema ainda maior e mais complexo do que o apresentado anteriormente, a mesma atividade gastaria muito mais tempo.

Por sorte você está trabalhando com uma linguagem que possui elementos reflexivos disponíveis para a programação. Sendo assim, seu programa conhece sua própria estrutura e pode realizar operações sobre essa. Seu programa sabe todas as classes e objetos que foram criados, e lhe permite utilizar informações a cerca da estrutura das classes e dos objetos. Dessa forma basta recuperar os dados necessários para salvar a estrutura da classe no formato desejado e em seguida preencher com os dados dos objetos. Nesse caso você pode definir uma função que realiza esse procedimento para cada classe e seus objetos encontrados no sistema via reflexão.

Nesse exemplo a reflexão permitiu que o programador tivesse o conhecimento e possibilidade de utilizar de aspectos internos do próprio programa, como o nome das classes e seus atributos para realizar uma outra programação – salvar os dados dos objetos. Além disso essa implementação serviria para qualquer tamanho de sistema, uma vez ele foi tratado de maneira genérica e independe do contexto em que as classes foram criadas.

1.1 Causalidade

Uma vez tendo sido compreendida a ideia básica por de trás da reflexão de software, faz-se interessante discutir brevemente um conceito essencial e caracterizador da reflexão. Esse conceito

refere-se é a causalidade em sistemas computacionais reflexivos, que em linhas gerais aborda a conexão entre os elementos recursivos e os elementos comuns da programação.

Caso um sistema computacional tenha somente a habilidade de conhecer a si próprio, esse não deve ser considerado reflexivo, mas sim deve ser visto como um sistema introspectivo. Sistemas reflexivos devem permitir que alterações sejam realizadas em seus elementos reflexivos e que por sua vez sejam passíveis de impactar os componentes mais comuns do sistema.

Ao tratar de reflexão deve-se entender que existe uma divisão conceitual do sistema em duas partes. Uma delas trata das estruturas comuns criadas voltadas ao domínio. Já a outra refere-se aos elementos que permitem a reflexão. Um sistema verdadeiramente reflexivo deve permitir que haja alguma relação entre essas duas divisões, visto que é a partir dessa relação que pode-se realizar a metaprogramação.

Considerando o exemplo hipotético apresentado na seção anterior, ao alterar uma classe criada para o domínio do problema, você está alterando também o elemento reflexivo conectado a ela. Nesse caso se criarmos uma classe com dois atributos, o array de atributos hipotético dessa classe deve ser capaz de manter registro desse aspecto. Agora se você alterar o array de atributos diretamente adicionando um nome de atributo nele, o sistema reflexivo ideal irá adicionar esse novo atributo a estrutura da própria classe.

Essa ideia pode parecer desafiador em um primeiro momento, todavia é através dessa ligação que pode-se, por exemplo modificar elementos do seu programa em tempo real conforme necessidade, o que torna o software muito mais flexível.

1.2 Reflexão e orientação a objetos

A reflexão é algo que não limita-se a um único modelo computacional. É possível encontrar implementações de linguagens de programação de diferentes tipos de paradigmas que empregam em algum nível recursos de reflexão. Sejam implementações dedutivas ou imperativas, procedurais ou funcionais, é possível encontrar exemplos de linguagens com características reflexivas.

Entretanto é no contexto da Orientação a Objetos que pode-se observar uma maior aplicação e um maior destaque da reflexão. Classes e objetos são excelentes estruturas para representar um sistema inteiro, armazenar dados pertinentes e permitir a sua manipulação. Geralmente faz-se uso dessas características para abstrair elementos do domínio do problema (que muitas vezes é baseado em aspectos da vida real) e programar o sistema.

Ao tratar da reflexão é possível fazer uso dessas mesmas características para abstrair outro elemento do seu programa, que na verdade é ele próprio. É possível trabalhar com classes e objetos para definir e representar a própria estrutura do programa e do ambiente no qual está inserido. Além disso, devido ao fato da linguagem já suportar operações sobre objetos e classes naturalmente, o desenvolvedor pode facilmente programar essas estruturas que referem-se aos aspectos reflexivos do código.

O próprio conceito de causalidade torna-se muito mais fácil de ser implementado, visto que naturalmente em linguagens orientadas a objetos têm-se presente a programação através da interação entre objetos. Nesse caso a interação ocorre entre objetos encarregados da parte reflexiva do software, com objetos criados para o sistema em si.

De maneira resumida, é possível tratar os aspectos reflexivos de maneira muito similar ao que já é feito quando trabalha-se com classes e objetos para resolver problemas de um determinado domínio. Dessa forma é muito mais fácil implementar recursos reflexivos nas linguagens orientadas a objetos, bem como é muito mais prático fazer o seu uso.

1.3 Vantagens do uso da reflexão

A reflexão é um recurso que se bem empregado, é capaz de garantir diversas vantagens ao processo de desenvolvimento de um programa. Conforme fora possível observar no exemplo hipotético trabalhado no inicio da seção 1, a reflexão permite alcançar um nível mais alto de

generalização dos problemas através da flexibilidade atingida pelo conhecimento e manipulação da sua estrutura interna.

Além disso é possível utilizar desse recurso para criar certos programas com uma maior facilidade. Imagine um programa de depuração onde as informações necessárias sobre o próprio programa já são suportadas e oferecidas pela própria linguagem, onde ainda é possível realizar uma programação mais avançada utilizando-se desses recursos para construir uma ferramenta concisa.

Levando em consideração as vantagens supracitadas, é também possível concluir que a reflexão é um caminho para atingir a construção de códigos reutilizáveis, permitindo que bibliotecas complexas possam ser desenvolvidas para serem aplicadas nos mais diversos contextos.

2. Python e Reflexão

Python é uma linguagem de programação amplamente conhecida altamente, uma vez que pode ser considerada fácil para aprender, e muito flexível para ser aplicada nos mais diversos contextos. A seção atual pretende demonstrar alguns dos recursos reflexivos presentes nativamente em Python.

Entretanto, antes de partir diretamente para as discussões desses recursos, faz-se interessante apresentar algumas características da linguagem e de sua principal implementação (Cpython) que está sendo usada nesse trabalho. Logo abaixo é possível conferir os dois tópicos considerados pelo autor como essenciais para a reflexão em Python.

- Interpretada: Linguagens Interpretadas possuem uma maior facilidade para aplicar os conceitos de reflexão, uma vez que seu ambiente executa instrução por instrução, facilitando que mudanças sejam feitas em tempo de execução;

- Estruturação e Objetos: Como será possível ver na subseção 2.1, a linguagem Python naturalmente está estruturada usando objetos e classes, e como já fora abordado, a orientação a objetos é uma grande facilitadora da reflexão de software;

Tendo sido observado esses aspectos gerais da linguagem, é possível dar sequência ao trabalho apontando algumas das principais características reflexivas em Python. Nas seções abaixo é possível compreender como alguns recursos reflexivos estão presentes na linguagem. A demonstração prática desses tópicos podem ser observadas a partir da seção 3 deste trabalho.

2.1 Auto-representação

A Auto-representação consiste na capacidade do sistema incorporar estruturas que representam a si próprio. É dessa forma que sistemas reflexivos são capazes de guardar informações referentes ao próprio programa, bem como realizar operações sobre os aspectos internos do sistema.

Python é uma linguagem onde tudo pode ser considerado um objeto. Toda a estruturação interna da linguagem é construída utilizando-se de conceitos da orientação a objetos. Para dar um exemplo desse aspecto, em Python o tipo de dado Int não é primitivo como em outras linguagens (C e C++), mas sim um objeto de uma classe, sendo essa usada para definir como dados inteiros devem se comportar no sistema.

Observado isso, é importante apresentar que Python necessita definir algumas classes “superiores” que vão além do nível de domínio para permitir que um programa seja criado. Essas classes são denominadas de metaclasses e são utilizadas para definir como as estruturas de um programa devem se comportar. É nas metaclasses por exemplo que pode-se encontrar informações especificando como um objeto deve ser exibido em um console de terminal, como dois números inteiros devem ser somados, como um valor desse ser avaliado logicamente, e como novos objetos devem ser criados no sistema.

O ponto principal que pretende-se discutir aqui é que através dessa organização da linguagem Python o sistema é capaz de representar a si mesmo e modificar o seu funcionamento interno alcançando assim, a reflexão.

Um exemplo muito simples é a capacidade do programador alterar como objetos serão exibidos no console do terminal. Todo objeto criado pelo programador irá possuir um método especial que sua classe herdou de uma metaclasse definindo como esse deve ser exibido. Todavia através do conceito de sobrescrita de métodos, é possível alterar esse aspecto e modificar conforme a necessidade do programador como o objeto será exibido. Perceba que nesse caso o desenvolvedor está “programando a programação do sistema”, alterando aspectos do funcionamento interno da linguagem e de seu ambiente.

2.2 Metaclasses

As metaclasses são as estruturas básicas do ambiente de programação Python. São elas que definem como um programa deve se comportar, e criam objetos que aplicam esses aspectos ao decorrer da execução do programa pelo interpretador.

Existem algumas metaclasses na linguagem, todavia as principais são type e object. Todo objeto é criado baseado em um objeto denominado object que por sua vez é definido pela metaclasse de mesmo nome. Já todo tipo de dado em Python é criado a partir de um objeto type, criado a partir da metaclasse de mesmo nome. São nessas duas metaclasses que pode-se encontrar as principais definições do programa, como metadados e métodos especiais para definir ações no sistema. São a partir delas que Python implementa por exemplo todos os tipo de dados e seus aspectos mais básicos, como endereço de memória, criação de novos objetos, acesso aos atributos dos objetos etc.

Devido ao fato desse conteúdo ser extremamente denso, e uma vez observado aspecto introdutório do trabalho, logo abaixo estão elencados alguns métodos definidos por essas metaclasses que definem os comportamentos do sistema. (No que tange aos dados e atributos presentes nessas estruturas, você pode encontrar mais informações na próxima subseção). Caso deseje ter mais conhecimento sobre esses aspectos internos da linguagem, pesquise diretamente na documentação da linguagem – link direto nas referências do trabalho.

- __new__(): Cria novos objetos de uma determinada classe passada por parâmetro;

- __del__(): Define algum procedimento para ser executado quando um objeto está prestes a ser apagado;

- __init__(): Após um objeto ser inicializado, o método __new__() retorna uma referência para esse objeto sendo ela chamada de self. Então faz-se uso do método __init__() para inicializar os valores de atributos do objeto, bem como iniciar algum eventual procedimento;

- __str__(): Define como um objeto deve ser representado quando esse estiver para ser exibido no formato de string. Pode ser usado para definir por exemplo como um objeto deve ser exibido quando for usado em conjunto da função print();

- __bool__(): Implementa a lógica de como um objeto deve ser avaliado logicamente;

- __eq__(): Implementa a lógica necessária para avaliar se dois objetos são iguais. É chamado quando desenvolve-se alguma expressão que utiliza o operador “==”;

- __call__(): Define o procedimento a ser executado quando o objeto for invocado. Permite que o programador invoque os objetos como se fossem funções;

Um aspecto interessante de possuir conhecimento é que o objeto type também herda de object, dessa forma object pode ser tido como topo da hierarquia de classes em Python. Todo elemento criado em python irá herdar dados e métodos dessa metaclasse.

Todavia, caso o programador deseje alterar o funcionamento interno dos objetos em Python, é possível definir uma metaclasse própria e fazer com que todos os outros objetos herdem a partir dela. Esse recurso é muito poderoso, uma vez que permite ao desenvolvedor modificar aspectos internos do funcionamento do ambiente de programação da linguagem e portanto deve ser trabalhado com cuidado, visando evitar erros que comprometam a operacionalidade do sistema.

2.3 Metadados

O conceito de metadados refere-se aos dados que tratam sobre algum aspecto do sistema em si, e não necessariamente do domínio em que esse está inserido. Um exemplo muito comum seria ter acesso ao nome de todas as variáveis declaradas no programa. Esse tipo de dados é muito importante, pois são eles que podem ser usados para representar o sistema, e posteriormente serem usados para alguma programação.

Como em Python tudo é um objeto e já fora observado que esses são definidos por metaclasses, praticamente todos os metadados do sistema podem ser encontrados, acessados e manipulados como atributos nos objetos criados a partir das metaclasses da linguagem. Como ocorre na subseção anterior, esse aspecto da linguagem também é consideravelmente denso, e pretende-se apresentar apenas uma visão inicial da reflexão em Python. Por tanto logo em seguida é possível observar alguns metadados comuns a maioria das classes e que podem ser usados pelos programadores. Para informações mais detalhadas, segue-se a mesma recomendação anteriormente dada - pesquise diretamente na documentação da linguagem.

Metadados de definições (funções, métodos, classes e módulos)

- __name__: Refere-se ao nome da definição. Curiosidade: o nome do módulo padrão em que um programa Python comum é executado chama-se “__main__”;

- __annotations__: Um dicionário com as anotações (recurso descrito posteriormente);

Metadados de classes

- __bases__: Uma tupla contendo todas as superclasses diretas da classe atual;

Metadados de objetos

- __class__: Classe a qual o objeto é instância;

- __dict__: Pode ser usado para obter-se um dicionário com os atributos graváveis do objeto e seus valores;

Metadados de funções e métodos

- __doc__: String de documentação;

- __defaults__: Dicionário com os valores padrões definidos para os parâmetros pela função/método;

2.4 Alteração de código em tempo de execução

Outra característica reflexiva interessante suportada pela linguagem Python é a capacidade de alterar códigos em tempo de execução. Isso ocorre graças a implementação da linguagem ser interpretada e de sua organização interna consistir em objetos criados a partir de metaclasses.

Considerando que a linguagem irá executar comando por comando, e que todas as definições de um programa podem ser encontradas em objetos que por natureza podem ser passíveis de sofrerem modificações, basta que o desenvolvedor altere os metadados e os métodos herdados das metaclasses para modificar os objetos do sistema.

Sendo assim é possível por exemplo criar uma classe com nenhum atributo, nem método definidos pelo usuário para funcionar como um esqueleto de classe, contendo apenas o essencial herdado por padrão da metaclasse object, e em seguida alterar essa classe adicionando atributos e métodos conforme necessário em tempo de execução.

Exemplos dessa prática são apresentados a partir da seção 3. Nessa subseção o intuito é apenas apresentar a ideia de alterar códigos durante o tempo de execução e como essa prática pode ser entendida no âmbito da linguagem Python.

2.6 Outros recursos reflexivos interessantes de se conhecer

Logo em seguida são apresentados alguns outros recursos interessantes que permitem reflexão em Python. Sua apresentação é dada de maneira simples visando uma rápida introdução aos diferentes recursos. Exemplos práticos de códigos reflexivos desses aspectos podem ser observados a partir da seção 3 deste trabalho.

2.6.1 Anotações

Versões mais recentes da linguagem permitem que o desenvolvedor realize algumas anotações especiais em seus códigos. Similar ao que acontece com o uso de comentários, essas notações servem para agregar alguma informação à algum elemento do seu programa, entretanto esse recurso é mais poderoso do que comentários comuns. Anotações podem ser usadas e acessadas em tempo de execução pelo programa, funcionando como metadados que descrevem algum elemento do seu código.

Atualmente as anotações podem ser aplicadas no contextos de funções/métodos. A comunidade responsável pelos padrões da linguagem recomendam que esse tipo de recurso seja utilizado especialmente para informar o tipo de dado de variáveis, parâmetros de funções e de seus valores de retorno.

As anotações podem ser acessadas a qualquer instante através do atributo __annotation__. Os metadados serão retornados em forma de um dicionário, onde a chave corresponde ao dado anotado e o valor à anotação.

2.6.2 Funções eval() e exec()

Essas três são funções built-in da linguagem utilizadas para avaliar, executar e compilar códigos através de uma string python. Através dessas funções é possível executar código python de maneira dinâmica e alterar o funcionamento de um programa em tempo de execução. Logo abaixo estão listadas as funções, bem como as sua principal funcionalidade.

- eval(): Recebe uma string como parâmetro, sendo que essa deve representar uma expressão python. Ao ser chamada, a função irá avaliar a string como se fosse um expressão python definida no seu programa;

- exec(): Similarmente a função anterior recebe uma string como parâmetro, mas essa deve representar um comando da linguagem python. Ao ser chamada, a função irá avaliar a string como se fosse um comando python definida no seu programa. Com essa função pode-se, por exemplo, solicitar ao usuário que informe um nome de variável e depois criar essa variável em seu programa;

Existe ainda uma terceira função que se enquadraria nesse contexto, sendo ela denominada compile(). Sua função é compilar uma string que contém código Python para código objeto, ou para a forma de Abstract Syntax Tree, sendo que essa forma refere-se a um formato utilizado pela linguagem antes de transformar o código para bytecode. Seu uso é mais complexo e portanto não é abordado no trabalho. Fica aqui registrada uma dica de tópico para aprofundamento para aqueles que desejam se aprofundar em assuntos como compilação e sintaxe e máquinas virtuais.

2.6.3 Funções para acesso à metadados

Alguns metadados podem ser acessados através da invocação de funções built-in da linguagem Python, facilitando dessa forma o acesso e impedindo que os dados sejam alterados sem querer, visto que essas funções permitem apenas um acesso “a leitura” dos dados. Logo abaixo estão elencadas algumas dessas funções consideradas mais interessantes e com potencial de aplicabilidade.

- type(): Retorna o tipo do objeto passado via parâmetro a função. O tipo do objeto por sua vez também é um objeto da metaclasse type e contém metadados que podem ser uteis ao programador. Observação: o tipo de um objeto é imutável e portanto não pode ser alterado.

- id(): Retorna um valor único identificador do objeto informado como parâmetro no sistema. Em CPython esse valor corresponde ao endereço de memória do objeto.

- isinstance(): Verifica se um objeto informado é de um determinado tipo, podendo inclusive ser uma instância indireta desse tipo. Retorna um valor logico indicado a relação de instância.

- issubclass(): Verifica se uma classe é uma subclasse direta de outra classe informada.

- dir(): Se usada sem parâmetros, devolve uma lista de nomes globais. Caso seja informado um objeto, a função irá retornar uma lista contendo todos os atributos daquele objeto.

- callable(): Retorna um valor lógico correspondendo se o objeto informado pode ser chamado (como ocorre em funções). Essa função verifica se o objeto possui o método __call()__ definido.

- hasattr(): Verifica se um objeto possui um determinado atributo informado sem a necessidade de usar estruturas de tratamento de exceções.

- globals(): Retorna o nome das variáveis globais do pacote no qual a função foi invocada.

- object(): Devolve um objeto da metaclasse object que serve como base para todas as outras classes da linguagem. Esse objeto retornado não possui nenhuma funcionalidade implementada e não pode ser alterado, nem diretamente, nem indiretamente tentando acessar os metadados (pois eles não existem nesse objeto!).

2.6.4 Recomendações de Tópicos sobre Reflexão para aprofundamento

Visando manter a simplicidade do trabalho, mas almejando auxiliar os leitores a prosseguirem os seus estudos em reflexão usando Python, a subseção atual apresenta alguns tópicos interessantes e que valem a pena serem analisados e estudados para adquirir uma maior capacidade de utilizar recursos reflexivos, bem como avançar o nível dos programas desenvolvidos. Confira esses tópicos logo em sequência.

- Decoradores: Usado para aprimorar a chamada de funções;
- Iteradores: Usado para definir objetos contêineres passíveis de serem iterados;
- Geradores: Usado para permitir que objetos sejam iterados sem a necessidade de um Iterador;
- Corrotinas: Usado para tratar de eventos síncronos e assíncronos;
- Frames: Relacionado ao código Python e sua execução;
- Código Objeto e AST: Relacionados ao modo como Python transforma seu código em código binário;
- PyObject: Base para a construção da linguagem. É como Objetos são representados na linguagem que implementa o interpretador Python.

3. Aplicação da reflexão em Python

A atual seção do trabalho pretende apresentar alguns exemplos de códigos reflexivos em Python, bem como o resultado de sua execução para tornar o conhecimento até então discutido mais conciso.

Em um primeiro instante são apresentados alguns aspectos mais simples que foram abordados na seções anteriores, para que o leitor possa compreender os aspectos mais básicos da programação reflexiva. Logo após esse momento é possível observar alguns códigos mais elaborados que objetivam apresentar possíveis aplicações práticas no contexto de desenvolvimento de software para os diferentes aspectos reflexivos.

Todos os códigos podem ser observados facilmente no repositório do github disponibilizado no link: https://github.com/Lima001/BCC-POO-II no diretório denominado “Reflexão/Códigos”. Além disso todos os resultados em forma de *print* inseridos na atual seção poderão ser conferidos no mesmo repositório no diretório denominado “Reflexão/Prints”.

3.1 Demonstrações simples de reflexão

Como já fora dito, esse primeiro momento pretende apresentar alguns exemplos mais simples de programação usando recursos de reflexão. Logo após um breve comentário a cerca do propósito do código e dos resultados obtidos, é possível observar a exibição de cada código e de seu resultado de execução.

3.1.1 Exemplo 1 – Modificando o comportamento padrão de objetos

Nesse exemplo o principal objetivo é apresentar como é possível sobrescrever métodos definidos pelas metaclasses herdadas pela classe definida pelo usuário para alterar o comportamento dos objetos das classes em algumas situações. Como é possível observar, foi possível implementar

como um objeto deve se comportar quando usado em conjunto do operador de igualdade, quando for representado como string e quando for chamado similarmente a uma função.

Esse é o exemplo mais simples, uma vez que muitos desses aspectos são comumente utilizados pelos desenvolvedores para adicionar funcionalidades aos objetos de suas classes e definir como esses devem se comportar ao decorrer do programa.

exemplo1.py > ...

```python
class Exemplo:

    # Método __init__ é usado para inicializar um objeto.
    # É invocado após a metaclasse invocar o método __new__()
    def __init__(self, atr1: int, atr2: int):
        self.atr1 = atr1
        self.atr2 = atr2

    # Método chamado quando usamos o operador ==
    def __eq__(self, obj):
        '''
            Usado para verificar se dois objetos da classe Exemplo são iguais.
            Verifica se os respectivos atributos dos objetos são iguais, retornando
            verdadeiro se for o caso, e falso caso contrário.
        '''
        return (self.atr1 == obj.atr1 and self.atr2 == obj.atr2)

    # Método chamado quando queremos representar um objeto em formato de string
    def __str__(self):
        return f"at1: {self.atr1} at2: {self.atr2}"

    # Método invocado quando realizamos uma chamada ao objeto usando nome_objeto()
    def __call__(self):
        return self.atr1 + self.atr2


# Criação de objetos simples para o exemplo
obj1 = Exemplo(1,2)
obj2 = Exemplo(1,4)
obj3 = Exemplo(1,4)

# Uso do metadado __name__ para saber se o pacote que estamos usando é o padrão (chamado __main__)
# Quando importamos uma biblioteca em Python, o seu nome é diferente de __main__, e o código abaixo
# não seria executado. Esse recurso é interessante, pois permite que executemos códigos apenas quando
# queremos testar alguma coisa do próprio módulo
if __name__ == "__main__":
    # Uso dos métodos sobrescritos na classe Exemplo
    print(obj1 == obj2)
    print(obj2 == obj3)
    print(obj1, obj2, obj3, sep="\n")
    print(obj1(), obj2(), sep="\n")
```

Figura 1: Código Exemplo 1

```
False
True
at1: 1 at2: 2
at1: 1 at2: 4
at1: 1 at2: 4
3
5
```

Figura 2: Execução do Código do Exemplo 1

3.1.2 Exemplo 2 – Acessando e Exibindo alguns Metadados

O principal objetivo desse exemplo é demonstrar como o acesso aos metadados pode ser feito em Python, bem como apresentar como esses podem ser exibidos ao programador através do uso da função print(). Como resultado é possível observar os dados contidos nesses metadados. Um

dos aspectos muito interessantes está relacionado ao fato do desenvolvedor ser capaz de recuperar as strings de documentação de uma função.

exemplo2.py

```
'''
    Exemplo criado para demonstrar o acesso a metadados disponibilizados
    pelas classes e objetos em Python
'''

from exemplo1 import *

if __name__ == "__main__":
    # Acesso aos metadados de classes diretamente
    print(Exemplo.__init__.__annotations__)     # Anotações do método __init__()
    print(Exemplo.__eq__.__doc__)               # Documentação do método __eq__()
    print(Exemplo.__bases__)                    # Superclasses de Exemplo
    print()
    print(obj1.__dict__)                        # Dicionário de atributos do objeto obj1
    print(obj2.__dict__)                        # Dicionário de atributos do objeto obj2
    print()
    print(type(obj1), type(obj2), type(obj3), type(Exemplo))    # Tipo dos objetos usando a função type()
    print(id(obj1), id(obj2), id(obj3))                         # id dos objetos usando a função id()
```

Figura 3: Código do Exemplo 2

```
{'atr1': <class 'int'>, 'atr2': <class 'int'>}

            Usado para verificar se dois objetos da classe Exemplo são iguais.
            Verifica se os respectivos atributos dos objetos são iguais, retornando
            verdadeiro se for o caso, e falso caso contrário.

(<class 'object'>,)

{'atr1': 1, 'atr2': 2}
{'atr1': 1, 'atr2': 4}

<class 'exemplo1.Exemplo'> <class 'exemplo1.Exemplo'> <class 'exemplo1.Exemplo'> <class 'type'>
139945219510032 139945219510128 139945219510224
```

Figura 4: Execução do Código do Exemplo 2

3.1.3 Exemplo 3 – Modificando o programa em tempo de execução usando strings

Já o atual exemplo visa apresentar como é possível alterar um programa em tempo de execução, bem como utilizar um dos recursos reflexivos mais interessantes ofertados pela linguagem, as funções eval() e exec(). Com o resultado da execução desse código é possível observar que um usuário é capaz de definir um nome em tempo de execução para adicionar um atributo a estrutura de um objeto com essa mesma nomenclatura.

Além disso é possível observar a característica de causalidade do sistema reflexivo, uma vez que a alteração do metadado responsável por armazenar as informações dos atributos do objeto do exemplo é responsável por alterar o próprio objeto em si.

exemplo3.py > ...

```python
'''
    Exemplo criado para demonstrar como podemos alterar o código em tempo
    de execução do programa, criando novas variáveis usando um nome informado
    pelo próprio usuário.

    Além disso, nesse código modificamos a estrutura de objetos através da manipulação
    de metadados que controlam seus atributos. Nesse caso estamos acrescentando
    atributos que antes não eram definidos pela sua classe. Como medida de segurança,
    a classe original não é alterada pelo Python, mas o objeto contará com o novo atributo
    em sua estrutura.
'''

from exemplo1 import *

if __name__ == "__main__":
    # Solicitar o nome para uma variável
    nome_novo_atributo = input("Digite um nome para um novo atributo do obj1: ")

    # Uma string contendo uma expressão matemática
    valor = "(2+2) * 5"

    # Avaliamos a expressão numérica, fazendo com que valor seja 20
    valor = eval(valor)

    # Criamos um atributo para o obj1 com o nome de variável informado pelo usuário,
    # e com o valor referente a avaliação da expressão valor vista anteriormente
    exec(f"obj1.{nome_novo_atributo} = {valor}")

    # Modificamos diretamente um metadado do objeto responsável pelo controle dos
    # atributos e de seus valores. Nesse caso estamos acrescentando um atributo atr0
    # cujo valor será 11
    obj1.__dict__["atr0"] = 11

    # Imprimindo os atributos do objeto usando metadados
    print(obj1.__dict__)

    # Imprimindo o atributo adicionando via manipulação de metadados
    # Nesse caso podemos observar que existe causalidade entre os aspectos
    # reflexivos e não reflexivos da linguagem
    print(obj1.atr0)
```

Figura 5: Código do Exemplo 3

```
lima001@lima001-Inspiron-3442:~/Documentos/IFC/BCC/3º Semestre/Programação OO II/Avaliações/AV1/Códigos$
ocumentos/IFC/BCC/3º Semestre/Programação OO II/Avaliações/AV1/Códigos/exemplo3.py"
Digite um nome para um novo atributo do obj1: novo
{'atr1': 1, 'atr2': 2, 'novo': 20, 'atr0': 11}
11
lima001@lima001-Inspiron-3442:~/Documentos/IFC/BCC/3º Semestre/Programação OO II/Avaliações/AV1/Códigos$
ocumentos/IFC/BCC/3º Semestre/Programação OO II/Avaliações/AV1/Códigos/exemplo3.py"
Digite um nome para um novo atributo do obj1: outro
{'atr1': 1, 'atr2': 2, 'outro': 20, 'atr0': 11}
11
```

Figura 6: Duas Execuções do Código do Exemplo 3

3.1.4 Exemplo 4 – Como classes são definidas

O último exemplo tem como principal meta apresentar uma sintaxe diferente para a definição de classes em Python. Essa sintaxe é referente a maneira como a própria linguagem é capaz de criar essas estruturas quando um código de definição de classe comum é executado. Como resultado é possível observar que existe uma relação entre classes e metaclasses, bem como a origem de alguns metadados já conhecidos.

exemplo4.py > ...

```python
'''
    Exemplo criado para demonstrar como é possível criar classes na linguagem
    Python usando uma síntaxe utilizada pela própria estrutura da linguagem
'''

if __name__ == "__main__":
    # Criação de uma classe chamada NomeClasse, que herda de object e possui
    # o atributo atr1 definido como 10, e o método func() retornando 10.

    # Observe que usamos type() para criar essa classe.
    # Isso ocorre, pois as classes em python devem herdar da metaclasse type.
    # Essa é a síntaxe usada internamente pela linguagem quando criamos uma classe
    # da maneira usual. Todavia observe que esse exemplo é bem simplificado para
    # um fácil entendimento
    classe1 = type("NomeClasse", (object,), {"atr1": 10, "func": lambda f: 10})

    # Criação de um objeto da classe NomeClasse, cuja referência pode ser acessada via variável classe1
    obj1 = classe1()

    # Exibição de atributos, invocação de métodos e acesso a metadados
    print(obj1.atr1)
    print(obj1.func())
    print(obj1.__class__.__name__)
    print(type(obj1))
```

Figura 7: Código do Exemplo 4

```
10
10
NomeClasse
<class '__main__.NomeClasse'>
```

Figura 8: Resultado da Execução do Código do Exemplo 4

3.2 Apresentação de exemplos próprios aplicando reflexão para solução de problemas

A subseção atual traz como aspecto principal a apresentação de 3 códigos reflexivos que visam resolver problemas que podem ser parte do processo de desenvolvimento de software. Assim como ocorre na subseção anterior, têm-se primeiramente um breve comentário a cerca do propósito do código e dos resultados obtidos sendo esses seguidos da exibição de cada código e de seu resultado de execução .

3.2.1 Verificação de tipos para instanciação de Objetos

Nesse exemplo através do uso das anotações do método __init__() presente na classe MinhaClasse, é possível entender quais os tipos de dados que o desenvolvedor definiu para cada atributo dos objetos dessa classe. Fazendo-se uso de uma função auxiliar que abstrai o processo de criação de um objeto, é possível resgatar essa informação e verificar se o tipo de dado que será usado para criar o objeto bate com a definição do desenvolvedor.

Como resultado é possível observar que apenas objetos cujos atributos possuem os mesmos tipos de dados informados pelo desenvolvedor através de anotações de métodos são criados, sendo que em outros casos o interpretador exibe uma mensagem de erro para alertar aquele que estiver instanciando objetos.

Esse exemplo é bem interessante, uma vez que evita a possibilidade de objetos serem criados com atributos de tipos errados, aspecto que eventualmente pode causar um erro no código e finalizar com a execução de um sistema.

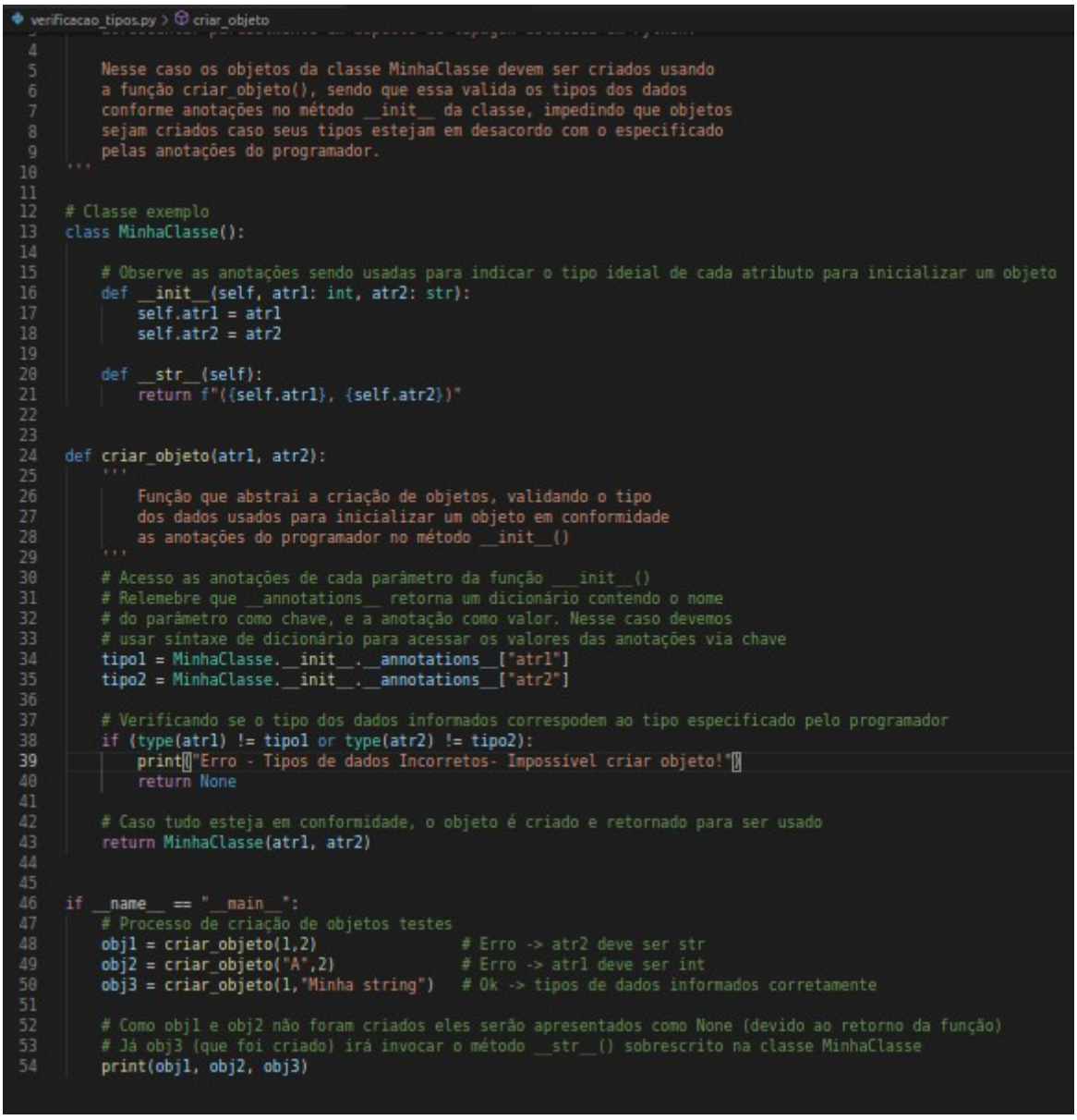

Figura 9: Código para Instanciação de Objetos verificando o tipo de Seus atributos

```
Erro - Tipos de dados Incorretos- Impossível criar objeto!
Erro - Tipos de dados Incorretos- Impossível criar objeto!
None None (1, Minha string)
```

Figura 10: Resultado da Execução do Código apresentado anteriormente

3.2.2 Armazenar dados e Recuperar dados usando Reflexão

Os códigos a seguir visam implementar de maneira simplificada o contexto descrito no início desse trabalho, quando uma situação hipotética de problema foi apresentada para permitir uma ideia inicial de reflexão e de sua aplicabilidade.

No caso desse exemplo existem dois códigos, onde o primeiro é responsável por criar classes e objetos e em seguida, usando reflexão salvar todos os dados interessantes a cerca desses elementos em uma lista. Essa lista de dados por sua vez é importada pelo segundo programa, sendo que esse é responsável por meio de reflexão em reconstruir o sistema do primeiro programa, criando cada classe e objeto “salvo” na lista.

Esse exemplo é bem interessante, pois caso seja trabalhando corretamente permite que o desenvolvedor utilize outros meios de salvar e recuperar dados. Imagine dados em um banco de dados, ou até mesmo em formato JSON que o programador precisa usar em seu programa. Sem reflexão seria necessário criar manualmente as classes e objetos para representar esses dados no seu programa, todavia usando os recursos de reflexão estudados nesse trabalho é possível automatizar essa tarefa e transferir esses dados para o formato desejado.

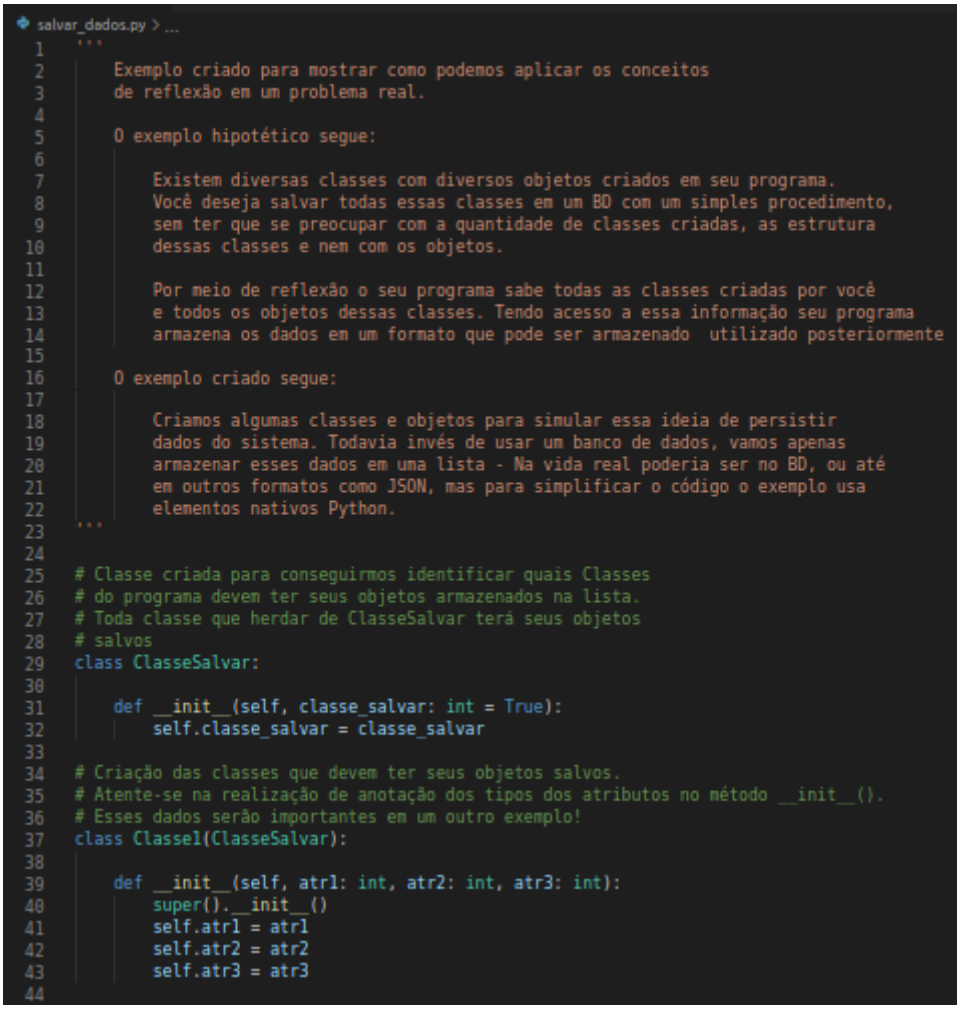

Figura 11: Código para salvar os Dados em uma Lista - Parte 1

salvar_dados.py > ...

```python
# Percorrer os nomes globais em procura das classes e objetos que queremos guardar
for i in dados_gerais:

    # Verificamos se o tipo do dado é do mesmo tipo que type, pois assim
    # conseguimos verificar se um dado nessa lista representa uma classe.
    # O tipo de uma classe é o mesmo tipo que type!
    #
    # Além disso fazemos outra verificação, pois podem existir classes que não
    # foram criadas pelo programador e que não devem ter seus objetos armazenados.
    # Sendo assim verificamos se a classe herda de ClasseSalvar (recurso criado especificamente
    # para esse propósito) acessando o metadado __bases__[0].__name__ (Nome da primeira superclasse)
    if type(i) == type(type) and "ClasseSalvar" == i.__bases__[0].__name__:
        # Pegamamos o nome da classe que queremos ter os objetos salvos e a anotação do método __init__()
        nome_classe = i.__name__
        atributos = i.__init__.__annotations__

        # Guardamos os dados anteriores seguindo o padrão especificado anteriormente
        lista_salvar.append([nome_classe, atributos])

    # Caso o dado não seja de uma classe, vamos ver se ele é de um objeto de alguma classe
    # que queremos salvar.
    else:
        # Verificamos se o objeto tem o atributo herdado de ClasseSalvar
        # cujo propósito é justamente identificar quais objetos de quais classes queremos guardar
        if hasattr(i, "classe_salvar"):
            # Pegamos o nome da classe a qual esse objeto é instância
            nome_classe = i.__class__.__name__

            # Procuramos na lista onde estamos salvando os objetos dessa classe
            for j in lista_salvar:
                if j[0] == nome_classe:
                    # Salvamos os atributos do objeto no local adequado seguindo o pdraõ
                    # de representação citado anteriormente
                    j.append(i.__dict__)

# Execução apenas para mostrar os dados que foram "salvos/persistidos" na lista_salvar
if __name__ == "__main__":
    for i in lista_salvar:
        # Exibir o nome da classe, seus atributos e tipos
        print(i[0], i[1])

        # Exibir os objetos salvos dessa classe
        for j in i[2:]:
            print(j)

        print()
```

Figura 13: Código para salvar os dados em uma lista - Parte 3

```
Classe1 {'atr1': <class 'int'>, 'atr2': <class 'int'>, 'atr3': <class 'int'>}
{'classe_salvar': True, 'atr1': 1, 'atr2': 2, 'atr3': 3}
{'classe_salvar': True, 'atr1': 3, 'atr2': 5, 'atr3': 7}
{'classe_salvar': True, 'atr1': 8, 'atr2': 10, 'atr3': 15}

Classe2 {'atr1': <class 'int'>, 'atr2': <class 'str'>, 'atr3': <class 'float'>}
{'classe_salvar': True, 'atr1': 1, 'atr2': 'Dado1', 'atr3': 7.6}
{'classe_salvar': True, 'atr1': 2, 'atr2': 'Dado2', 'atr3': 8.9}
{'classe_salvar': True, 'atr1': 3, 'atr2': 'Dado3', 'atr3': 5.6}

Classe3 {'atr1': <class 'str'>}
{'classe_salvar': True, 'atr1': 'I'}
{'classe_salvar': True, 'atr1': 'J'}
```

Figura 14: Execução do código anterior - Dados que serão passados para o programa 2 dentro de uma lista

recuperar_dados.py > ...

```python
'''
    Exemplo criado para recuperar as informações armazenadas
    no exemplo recuperar_dados.py

    Nesse exemplo, sem conhecer as classes do programa anterior,
    tendo apenas conhecimentos dos dados e do formato que esses estão
    armazenados, vamos recuperar as classes e objetos do programa anterior.

    Esse exemplo mostra como a reflexão poderia ser util para recuperar dados
    de uma fonte externa sem ter que o programador propriamente definir as
    classes usadas para criar os objetos.
'''

from salvar_dados import Classe1, Classe3, lista_salvar, ClasseSalvar

# Contador de objetos recuperados
count_obj = 0

# Vamos percorrer a lista de dados que recuperamos do outro programa
# Onde que cada iteração desse laço principal estaremos trabalhando
# com uma classe específica e os objetos dela que foram armazenados
for i in range(len(lista_salvar)):

    # Recuperação das informações importantes
    nome_classe = lista_salvar[i][0]
    tipos_dados = list(lista_salvar[i][1].values())
    atributos = list(lista_salvar[i][1].keys())
    objetos = lista_salvar[i][2:]

    # String contendo o código do cabeçalho de criação de uma classe em Python
    codigo_classe = f"class {nome_classe}(ClasseSalvar):\n"

    # Criação do método __init__() da classe recuperada da lista
    metodo_init = "\tdef __init__(self"

    # Adição dos atributos na definição do método __init__()
    for chave in atributos:
        metodo_init += f", {chave}"

    # Código de incialização da superclasse
    metodo_init += "):\n\t\tsuper().__init__()\n"
```

Figura 15: Código para recuperar os dados do programa 1 - Parte 1

```
47
48      # Caso queira ver a string de código para criação de uma classe, descomente a linha abaixo
49      #print(codigo_classe + metodo_init)
50      # Execução da string gerada para criação de uma classe recuperada da lista
51      exec(codigo_classe + metodo_init)
52
53      # Iteração para pegar as informações de cada objeto da classe que acabou de ser criada.
54      # Os dados desses objetos são advindos da lista de dados recuperada do outro programa
55      for obj in objetos:
56
57          # Definição de uma string para criação de uma variável que contera
58          # um objeto da classe criada anteriormente
59          criar_objeto = f"obj{count_obj} = {nome_classe}("
60
61          # Dados necessários para recuperar os dados do objeto a ser criado
62          tamanho = len(obj.values())
63          valores = list(obj.values())[1:]
64
65          # Percorrer os dados recuperados da lista sobre aquele objeto a ser criado
66          for j in range(tamanho-1):
67              # Adicionar os dados dos atributos do objeto, já realizando a sua conversão
68              # usando a informação de tipo resgatada também da lista de dados
69              criar_objeto += f"tipos_dados[{j}](valores[{j}])"
70
71              if j == tamanho-2:
72                  criar_objeto += ")"
73              else:
74                  criar_objeto += ", "
75
76          # Descomente a linha abaixo para ver como fica a string de criação de um objeto
77          #print(criar_objeto)
78
```

Figura 16: Código para recuperar os dados do programa 1 - Parte 2

```
79          # Somar 1 a quantidade de objetos - Essa informação é relevante, pois
80          # cada objeto estará associado a uma variável. Para tornar uma variável
81          # única e não ter conflito de nomes, usamos esse valor para componente do
82          # seu nome.
83          count_obj += 1
84
85          # Execução da string que cria um objeto
86          exec(criar_objeto)
87
88  # Exibição dos objetos recuperados da lista advinda do outro programa.
89  # Dica: Para deixar a exibição melhor, você pode reflexivamente implementar
90  # o método __str__() para as classes recuperadas e os dados de seus atributos
91  # armazenados na lista_salvar
92  print(obj0.atr1, obj0.atr2, obj0.atr3)
93  print(obj1.atr1, obj1.atr2, obj1.atr3)
94  print(obj2.atr1, obj2.atr2, obj2.atr3)
95  print(obj3.atr1, obj3.atr2, obj3.atr3)
96  print(obj4.atr1, obj4.atr2, obj4.atr3)
97  print(obj5.atr1, obj5.atr2, obj5.atr3)
98  print(obj6.atr1)
99  print(obj7.atr1)
100
101 # Criação de um novo objeto sem ter a definição da classe escrita ou importada pelo desenvolvedor no programa
102 obj8 = Classe3("A")
103
104 # Exibição do novo objeto
105 print(obj8.atr1)
```

Figura 17: Código para recuperar os dados do programa 1 - Parte 3

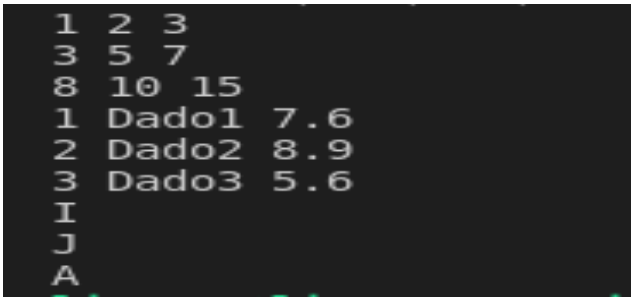

Figura 18: Exibição dos atributos dos objetos criado a partir dos dados advindos do programa 1, recuperados via reflexão

FINALIZANDO

Com o presente trabalho tendo sido apresentado, é possível concluir que o recurso de reflexão pode ser muito útil se bem empregado no processo de desenvolvimento de software, uma vez que permite atividades relacionadas a própria programação do sistema sejam automatizadas. Além disso é possível observar que Python é uma linguagem relativamente complexa e cheia de recursos, sendo que a reflexão é um muito presente em sua estrutura, observado o fato de que a linguagem toda está estruturada usando conceitos de Orientação a Objetos.

Ao longo do trabalho certas dificuldades foram encontradas sendo que as principais podem ser tidas como: a necessidade de aprofundar-se na estrutura da própria linguagem para compreender alguns recursos reflexivos, e o sentimento de falta de um material didático simples e direto sobre o tema, capaz de sanar as dúvidas mais facilmente do que ao pesquisar diretamente na documentação da linguagem.

De todo modo, julga-se que o desenvolvimento do trabalho foi muito interessante e recompensador, uma vez que permitiu a geração de um conhecimento sobre um recurso de muito potencial que a reflexão. Acrescenta-se aqui também a possibilidade de desenvolver outros trabalhos baseados nessa visão introdutória de reflexão em Python. Muitos tópicos mais complexos foram deixados de lados, e outros resumidos ao extremo. Python é uma linguagem extremamente versátil e sua estrutura é cheia de elementos inteligentes que valem a pena serem estudados e aprofundados.

REFERÊNCIAS

Reflection in Python. Disponível em: https://www.geeksforgeeks.org/reflection-in-python/#:~:text=Reflection%20refers%20to%20the%20ability,represents%20the%20type%20of%20obj

Python Reflection and Introspection. Por Jamie Bullock. Disponível em: https://betterprogramming.pub/python-reflection-and-introspection-97b348be54d8

Python Programming/Reflection. Disponível em: https://en.wikibooks.org/wiki/Python_Programming/Reflection

Reflective programming. Disponível em: https://en.wikipedia.org/wiki/Reflective_programming

Python 3.9.5 documentation. Disponível em: https://docs.python.org/3/

Data model. The Python Language Reference. Disponível em: https://docs.python.org/3/reference/datamodel.html

A note on reection in Python 1.5. Por Anders Andersen. Disponível em: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.50.9504&rep=rep1&type=pdf

Advanced Python: Metaprogramming. Por Farhad Malik. Disponível em: https://medium.com/fintechexplained/advanced-python-metaprogramming-980da1be0c7d

Metaprogramming in Python. Por Satwik Kansal. Disponível em: https://developer.ibm.com/technologies/analytics/tutorials/ba-metaprogramming-python/

Metaprogramming with Metaclasses in Python. Disponível em: https://www.geeksforgeeks.org/metaprogramming-metaclasses-python/

Expert Python Programming – Third Edition. Por Michal Jaworski e Tarek Ziadé