Programa “ConsoleRPN”

Programa em Python criado com o Visual Studio Code
para efetuar operações algébricas entre números inteiros,
uma calculadora funcionando com console usando RPN

PUC-Rio – CIV 2802 - Sistemas Gráficos para Engenharia
Luiz Fernando Martha & André Maués Brabo Pereira

Objetivos da Aula:

Apresentar os conceitos de programação da linguagem Python utilizando um exemplo de uma calculadora RPN (Reversed Polish Notation)

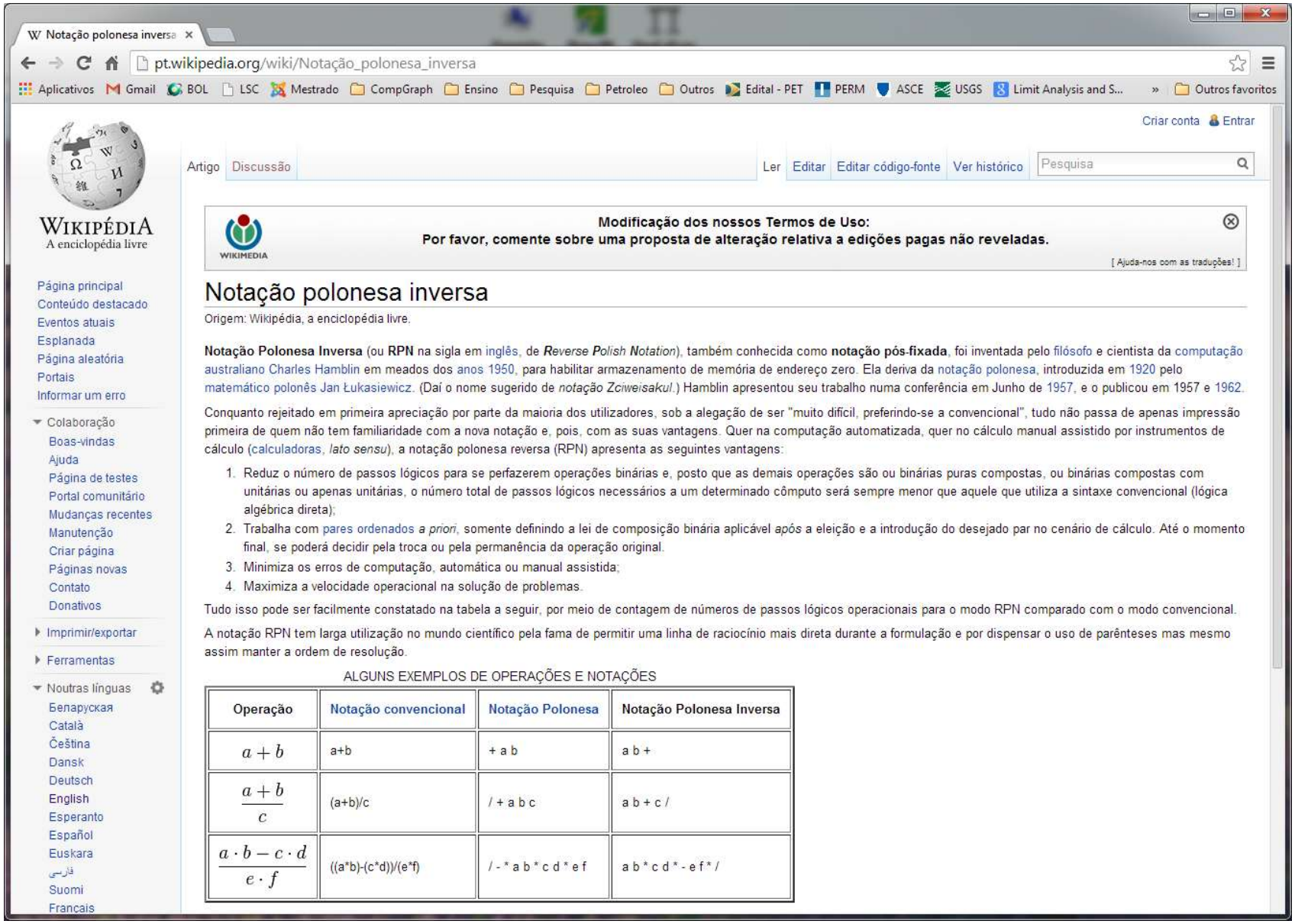
Conteúdo/Assuntos Abordados:

Introdução à linguagem de programação Python.
Introdução ao conceito de classes e objetos em Python.

Competências/Habilidades:

Conhecer os recursos básicos da linguagem Python.
Capacidade de criar um projeto console no Visual Studio Code.
Capacidade de entendimento de algoritmos e estruturas de dados.
Entendimento do uso de funções na linguagem Python.
O aluno deverá ser capaz de desenvolver aplicativos práticos.

O que significa uma calculadora RPN (Reserved Polish Notation)?



Notação polonesa inversa

pt.wikipedia.org/wiki/Notação_polonesa_inversa

Criar conta Entrar

Artigo Discussão Ler Editar Editar código-fonte Ver histórico Pesquisa

WIKIPÉDIA
A enciclopédia livre

Página principal
Conteúdo destacado
Eventos atuais
Esplanada
Página aleatória
Portais
Informar um erro

Colaboração
Boas-vindas
Ajuda
Página de testes
Portal comunitário
Mudanças recentes
Manutenção
Criar página
Páginas novas
Contato
Donativos

Imprimir/exportar

Ferramentas

Noutras línguas
Беларуская
Català
Čeština
Dansk
Deutsch
English
Esperanto
Español
Euskara
فارسی
Suomi
Français

Modificação dos nossos Termos de Uso:
Por favor, comente sobre uma proposta de alteração relativa a edições pagas não reveladas.
[Ajuda-nos com as traduções!]

Notação polonesa inversa

Origem: Wikipédia, a enciclopédia livre.

Notação Polonesa Inversa (ou **RPN** na sigla em inglês, de *Reverse Polish Notation*), também conhecida como **notação pós-fixada**, foi inventada pelo filósofo e cientista da computação australiano Charles Hamblin em meados dos anos 1950, para habilitar armazenamento de memória de endereço zero. Ela deriva da notação polonesa, introduzida em 1920 pelo matemático polonês Jan Łukasiewicz. (Daí o nome sugerido de *notação Zciweisakul*.) Hamblin apresentou seu trabalho numa conferência em Junho de 1957, e o publicou em 1957 e 1962.

Conquanto rejeitado em primeira apreciação por parte da maioria dos utilizadores, sob a alegação de ser "muito difícil, preferindo-se a convencional", tudo não passa de apenas impressão primeira de quem não tem familiaridade com a nova notação e, pois, com as suas vantagens. Quer na computação automatizada, quer no cálculo manual assistido por instrumentos de cálculo (calculadoras, *lato sensu*), a notação polonesa reversa (RPN) apresenta as seguintes vantagens:

1. Reduz o número de passos lógicos para se perfazerem operações binárias e, posto que as demais operações são ou binárias puras compostas, ou binárias compostas com unitárias ou apenas unitárias, o número total de passos lógicos necessários a um determinado cômputo será sempre menor que aquele que utiliza a sintaxe convencional (lógica algébrica direta);
2. Trabalha com pares ordenados *a priori*, somente definindo a lei de composição binária aplicável *após* a eleição e a introdução do desejado par no cenário de cálculo. Até o momento final, se poderá decidir pela troca ou pela permanência da operação original.
3. Minimiza os erros de computação, automática ou manual assistida;
4. Maximiza a velocidade operacional na solução de problemas.

Tudo isso pode ser facilmente constatado na tabela a seguir, por meio de contagem de números de passos lógicos operacionais para o modo RPN comparado com o modo convencional.

A notação RPN tem larga utilização no mundo científico pela fama de permitir uma linha de raciocínio mais direta durante a formulação e por dispensar o uso de parênteses mas mesmo assim manter a ordem de resolução.

ALGUNS EXEMPLOS DE OPERAÇÕES E NOTAÇÕES

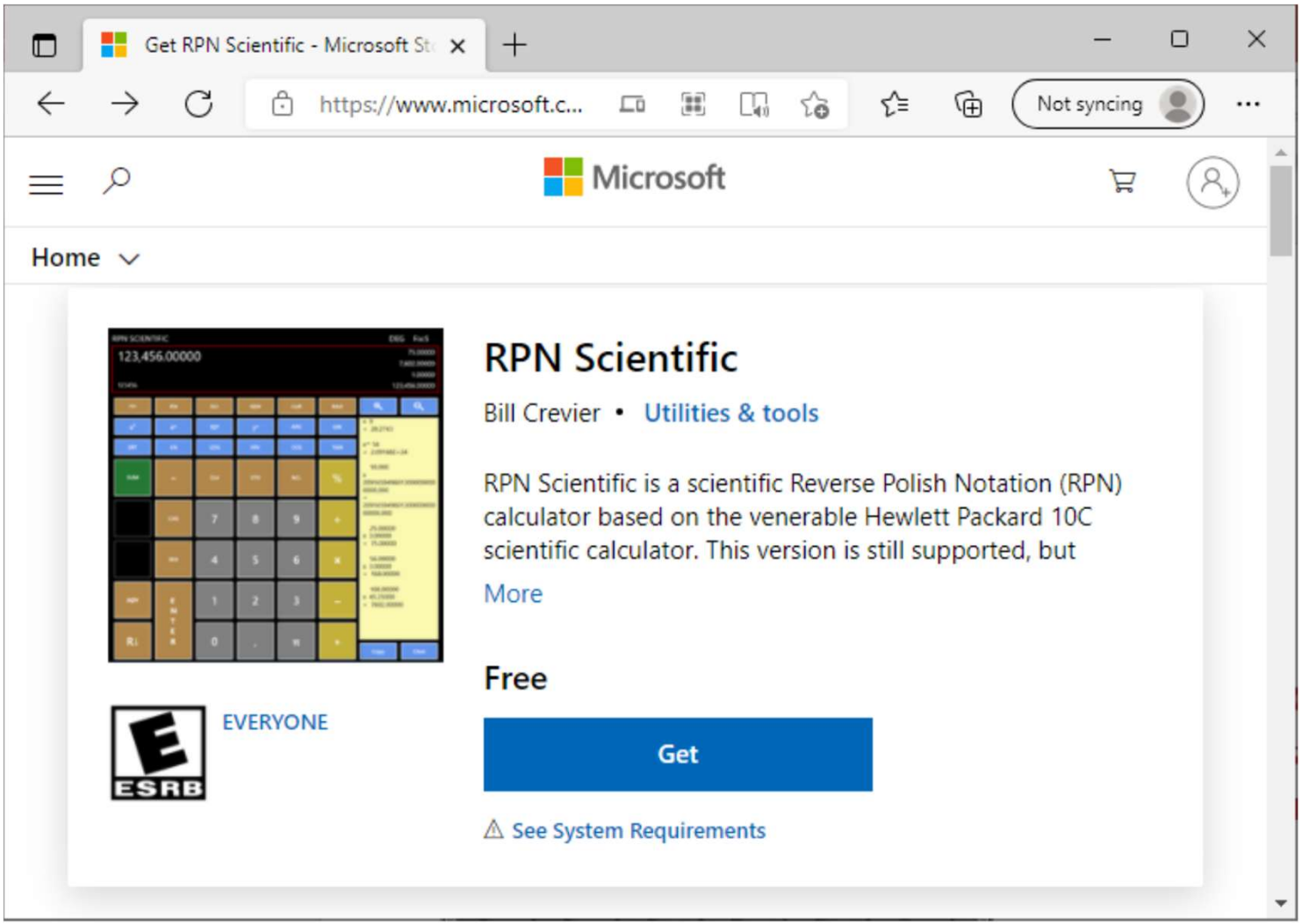
Operação	Notação convencional	Notação Polonesa	Notação Polonesa Inversa
$a + b$	a+b	+ a b	a b +
$\frac{a+b}{c}$	(a+b)/c	/ + a b c	a b + c /
$\frac{a \cdot b - c \cdot d}{e \cdot f}$	((a*b)-(c*d))/(e*f)	/ - * a b * c d * e f	a b * c d * - e f * /

Exemplo clássico de calculadora que utiliza a Notação Polonesa Reversa (RPN)

Como criar nossa própria calculadora utilizando uma linguagem de programação e uma interface gráfica?

Get free from Microsoft Store

https://www.microsoft.com/en-us/p/rpn-scientific/9wzdncrdcfx1?activetab=pivot:overviewtab



Estruturas de Dados

Qual a estrutura de dados mais adequada para o problema da calculadora RPN?

Firefox — W Estrutura de dados – Wikipédia, a encicl... — pt.wikipedia.org/wiki/Estrutura_de_dados — pilha

Criar conta | Entrar

WIKIPÉDIA
A enciclopédia livre

Artigo | Discussão | Ler | Editar | Editar código-fonte | Ver histórico | Pesquisa

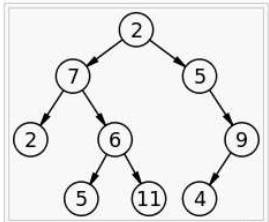
- Página principal
- Conteúdo destacado
- Eventos atuais
- Esplanada
- Página aleatória
- Portais
- Informar um erro
- Colaboração
  - Boas-vindas
  - Ajuda
  - Página de testes
  - Portal comunitário
  - Mudanças recentes
  - Manutenção
  - Criar página
  - Páginas novas
  - Contato
  - Donativos
- Imprimir/exportar
- Ferramentas
- Noutras línguas
  - العربية

Estrutura de dados

Origem: Wikipédia, a enciclopédia livre.

Na Ciência da computação, uma **estrutura de dados** é um modo particular de armazenamento e organização de dados em um computador de modo que possam ser usados eficientemente.[1] [2]

Diferentes tipos de estrutura de dados são adequadas a diferentes tipos de aplicação e algumas são altamente especializadas, destinando-se a algumas tarefas específicas. Por exemplo, as B-trees são particularmente indicadas para a implementação de bases de dados , enquanto que a implementação de compiladores geralmente requer o uso de tabela de dispersão para a busca de identificadores.



Uma árvore binária é uma estrutura de dados.

Estruturas de dados e algoritmos são temas fundamentais da ciência da computação, sendo utilizados nas mais diversas áreas do conhecimento e com os mais diferentes propósitos de aplicação. Sabe-se que algoritmos manipulam dados. Quando estes dados estão organizados (dispostos) de forma coerente, caracterizam uma forma, uma **estrutura de dados**. A organização e os métodos para manipular essa estrutura é que lhe conferem singularidade e diminuição do espaço ocupado pela memória RAM, além de tornar o código-fonte do prog simplificado.

As estruturas de dados são chamadas tipos de dados compostos que dividem-se em homogêneos (vetor (registros). As estruturas homogêneas são conjuntos de dados formados pelo mesmo tipo de dado primi heterogêneas são conjuntos de dados formados por tipos de dados primitivos diferentes (campos do regi estrutura. A escolha de uma estrutura de dados apropriada pode tornar um problema complicado em um simples. O estudo das estruturas de dados está em constante desenvolvimento (assim como o de algori existem certas estruturas clássicas que se comportam como padrões.

Índice [esconder]

Estruturas de dados clássicas [editar | editar código-fonte]

Vetores ou *arrays* [editar | editar código-fonte]

Ver artigo principal: Array

Vetores[PB], ou **vectores**[PE] ou *arrays* são estruturas de dados lineares e estáticas, isto é, são compostas por um número fixo (finito) de elementos de um determinado tipo de dados. O tempo de acesso aos elementos de um vetor é muito rápido, sendo considerado constante: o acesso aos elementos é feito pelo seu índice no vetor. Porém, a remoção de elementos pode ser custosa se não for desejável que haja espaços "vazios" no meio do vetor, pois nesse caso é necessário "arrastar" de uma posição todos os elementos depois do elemento removido.

Essa é uma estrutura muito recomendada para casos em que os dados armazenados não mudarão, ou pouco mudarão, através do tempo.

Lista [editar | editar código-fonte]

Ver artigo principal: Lista

Uma Lista é uma estrutura de dados linear. Uma lista ligada, também chamada de encadeada, é linear e dinâmica, é composta por nós que apontam para o próximo elemento da lista, o ultimo elemento apontará para nulo. Para compor uma lista encadeada, basta guardar seu primeiro elemento.

Fila [editar | editar código-fonte]

Ver artigo principal: FIFO

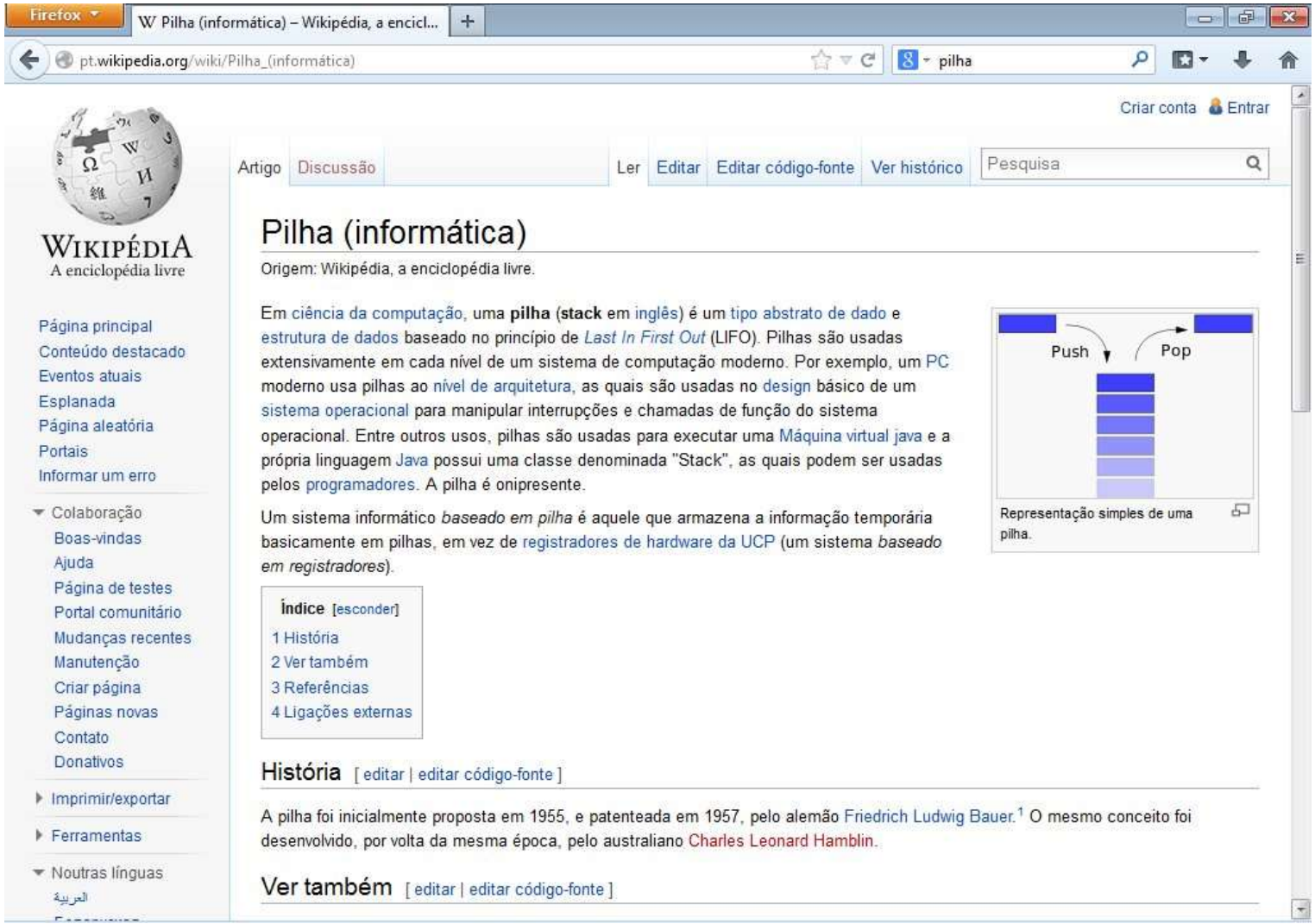
As filas são estruturas baseadas no princípio **FIFO** (*first in, first out*), em que os elementos que foram inseridos no início são os primeiros a serem removidos. Uma fila possui duas funções básicas: **ENQUEUE**, que adiciona um elemento ao final da fila, e **DEQUEUE**, que remove o elemento no início da fila. A operação **DEQUEUE** só pode ser aplicada se a fila não estiver vazia, causando um erro de **underflow** ou fila vazia se esta operação for realizada nesta situação.

Pilha [editar | editar código-fonte]

Ver artigo principal: LIFO

A pilha é uma estrutura de dados baseada no princípio LIFO (*last in, first out*), na qual os dados que foram inseridos primeiros na pilha serão os últimos a serem removidos. Existem duas funções que se aplicam a todas as pilhas: **PUSH**, que insere um dado no topo da pilha, e **POP**, que remove o item no topo da pilha.

A Estrutura de Dados Pilha (*Stack*)



Como implementar uma estrutura de dados de pilha na linguagem Python?

Programação Orientada a Objetos

Programação Orientada a Objetos (POO)

Classes e objetos

- Na POO, dados e funções estão intrinsicamente associados.

- Um **objeto** é uma entidade que tem dados, denominados **propriedades**, e funções, denominadas **métodos**, que manipulam suas propriedades.

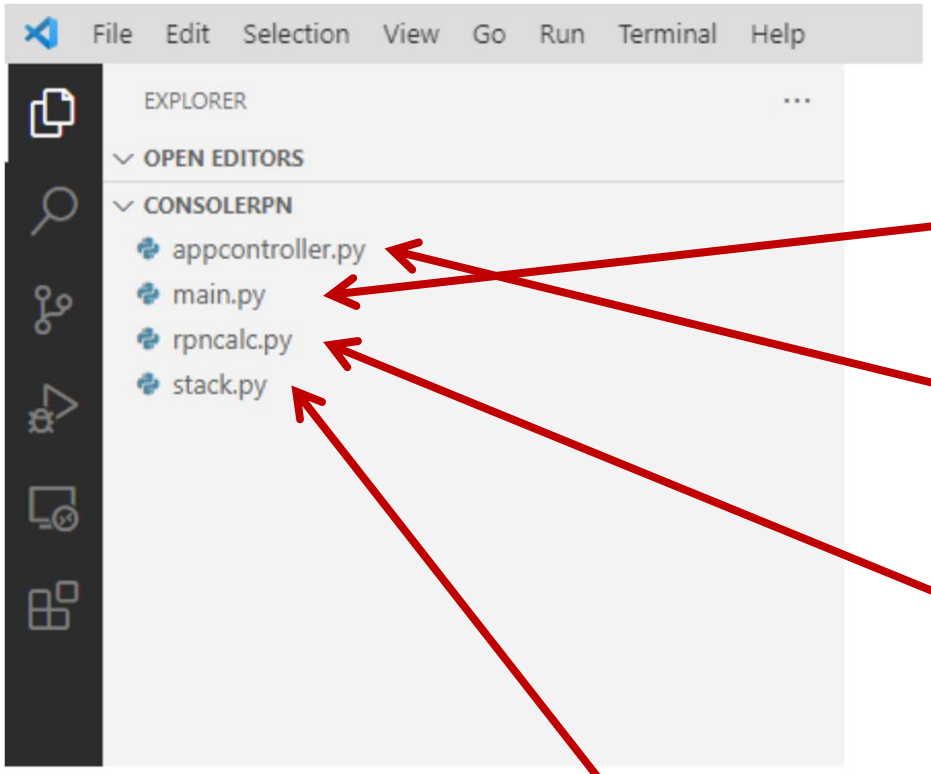
- Conceito de **encapsulamento**: na POO "pura", não existem variáveis desassociadas de objetos e a única maneira de modificar as propriedades (dados) de um objeto é através de seus métodos. É criada uma "cerca" que protege os dados de um objeto, só permitindo que esses sejam manipulados de maneira formal através de métodos. O encapsulamento é um dos motivos da robustez de programas orientados a objetos.

- Uma **classe** é um modelo para criação de objetos. Objetos são **instâncias** criadas de uma classe.

- A execução de um programa orientado a objetos se dá através da comunicação entre objetos através de chamadas a métodos.

Calculadora RPN de inteiros em Python – versão console

Interface do Visual Studio Code



dá partida da execução do programa

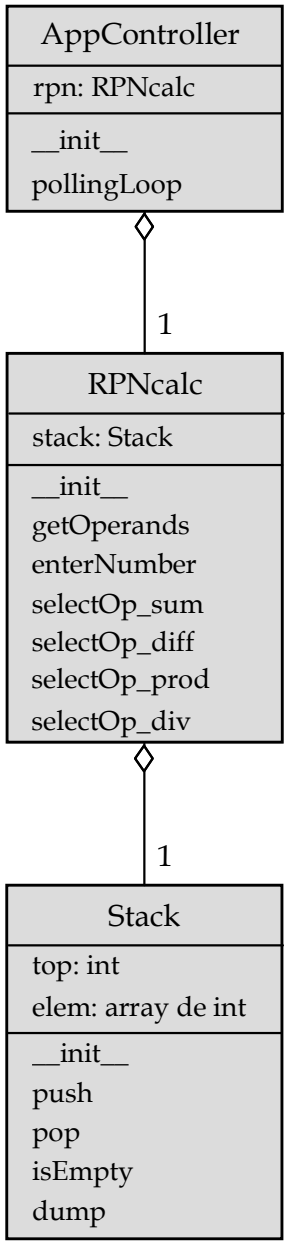
classe AppController para controlar a execução do programa

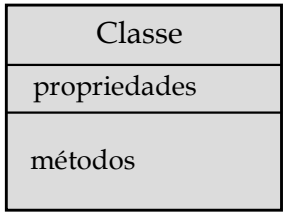
classe RPNCalc para calculadora RPN de inteiros

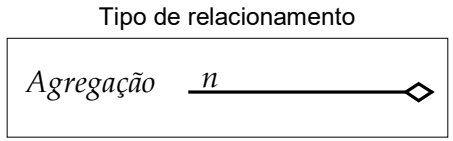
classe Stack para pilha

Diagrama UML de classes

UML: *Unified Modeling Language*







Pilha (Stack) de inteiros em Python utilizando “array”

Arquivo stack.py:

```
1  import numpy as np
2
3  class Stack:
4      def __init__(self):
5          self.top = -1
6          self.elem = np.zeros(50, dtype=int)
7
8      def push(self, _n):
9          if type(_n) == int:
10             self.top += 1
11             self.elem[self.top] = _n
12
13     def pop(self):
14         if self.top < 0:
15             return None
16         n = self.elem[self.top]
17         self.top -= 1
18         return n
19
20     def isEmpty(self):
21         return self.top < 0
22
23     def dump(self):
24         text = ""
25         for i in range(self.top,-1,-1):
26             text += f"{self.elem[i]}\n"
27         return text
28
```

classe Stack para pilha

__init__: método de inicialização
self: objeto corrente

propriedade top: índice do elemento do topo (–1: pilha vazia)

propriedade elem: array (vetor) de elementos da pilha de inteiros, inicializado com tamanho 50 e zerado

método push: insere um valor no topo da pilha

método pop: remove valor do topo da pilha e retorna esse valor

método isEmpty: retorna um booleano (true ou false) indicando se a pilha está vazia

índice i começa com valor top e termina com valor 0 (–1 não incluído), decrescendo de 1

método dump: retorna um texto com os valores dos elementos da pilha por linha

Arquivo rpncalc.py (objeto Stack como propriedade):

```python
from stack import *

class RPNcalc:
    def __init__(self):
        self.stack = Stack()

    def getOperands(self):
        if self.stack.isEmpty():
            print("Empty stack!\n")
            return False, 0, 0

        n1 = self.stack.pop()
        if self.stack.isEmpty():
            print("Two operands needed!\n")
            self.stack.push(n1)
            return False, 0, 0

        n2 = self.stack.pop()
        return True, int(n1), int(n2)

    def enterNumber(self, _n):
        self.stack.push(_n)
```

classe RPNcalc

propriedade: objeto da classe Stack

método enterNumber: insere valor no topo da pilha

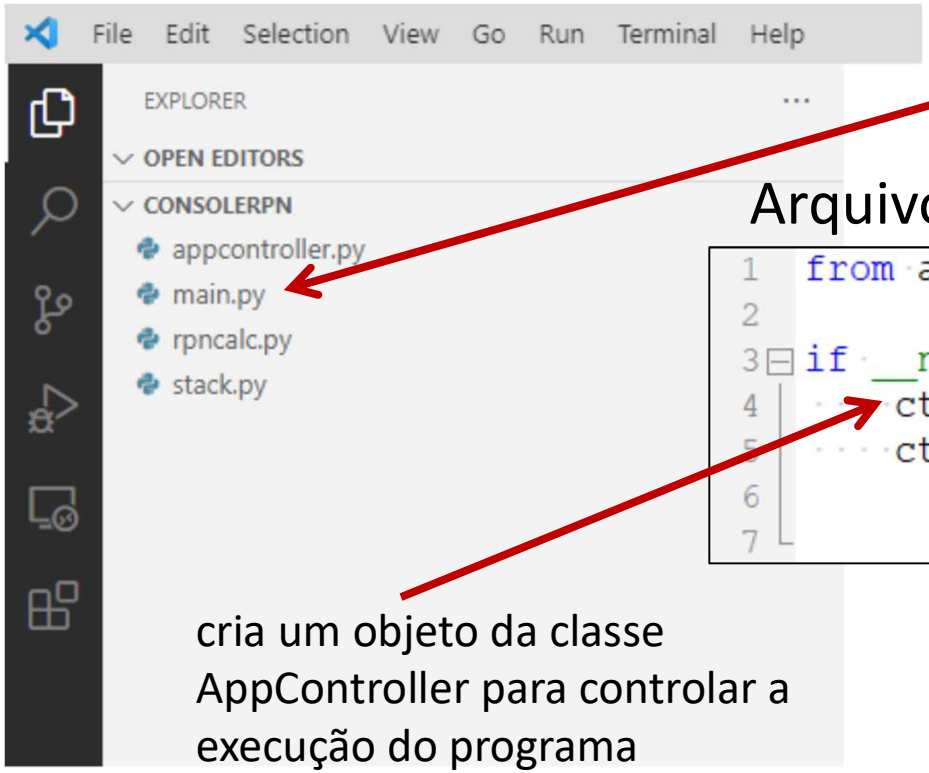
método getOperands: retorna (e desempilha) dois valores do topo da pilha

```python
    def selectOp_sum(self):
        check, n1, n2 = self.getOperands()
        if not check:
            return
        self.stack.push(n2+n1)

    def selectOp_diff(self):
        check, n1, n2 = self.getOperands()
        if not check:
            return
        self.stack.push(n2-n1)

    def selectOp_prod(self):
        check, n1, n2 = self.getOperands()
        if not check:
            return
        self.stack.push(n2*n1)

    def selectOp_div(self):
        check, n1, n2 = self.getOperands()
        if not check:
            return
        if n1 != 0:
            self.stack.push(int(n2/n1))
        else:
            print("Error: division by zero!\n")
```

métodos selectOp_sum, diff, prod, div: operações com dois valores removidos do topo da pilha e inserção do resultado no topo.

Calculadora RPN de inteiros em Python – versão console

Interface do Visual Studio Code



dá partida da execução do programa

Arquivo main.py:

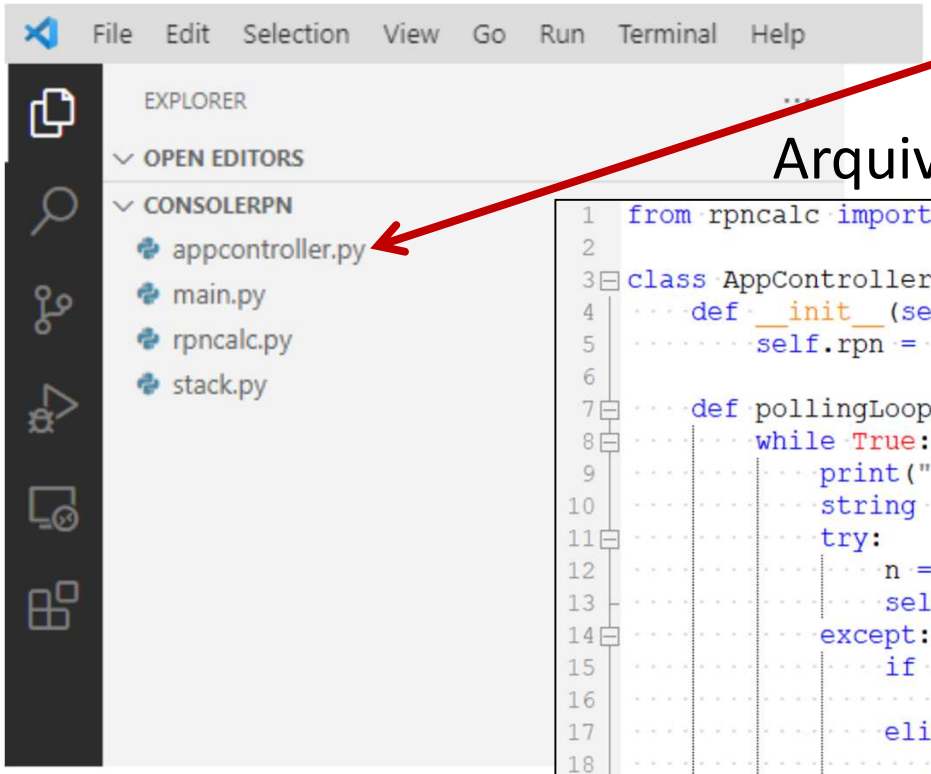
```
1  from appcontroller import AppController
2
3  if __name__ == '__main__':
4      ctrl = AppController()
5      ctrl.pollingLoop()
6
7
```

cria um objeto da classe AppController para controlar a execução do programa

chama o método pollingLoop do objeto da classe AppController

Calculadora RPN de inteiros em Python – versão console

Interface do Visual Studio Code



classe AppController para controlar a execução do programa

Arquivo AppController.py:

```
1  from rpncalc import *
2
3  class AppController:
4      def __init__(self):
5          self.rpn = RPNcalc()
6
7      def pollingLoop(self):
8          while True:
9              print("Enter a number or an operator ('q' to quit): ")
10             string = input()
11             try:
12                 n = int(string)
13                 self.rpn.enterNumber(n)
14             except:
15                 if string == 'q':
16                     break
17                 elif string == '+':
18                     self.rpn.selectOp_sum()
19                 elif string == '-':
20                     self.rpn.selectOp_diff()
21                 elif string == '*':
22                     self.rpn.selectOp_prod()
23                 elif string == '/':
24                     self.rpn.selectOp_div()
25                 else:
26                     print("Error: enter a valid operator!\n")
27             print("\n----------\n"+self.rpn.stack.dump()+"----------\n")
28
29
```

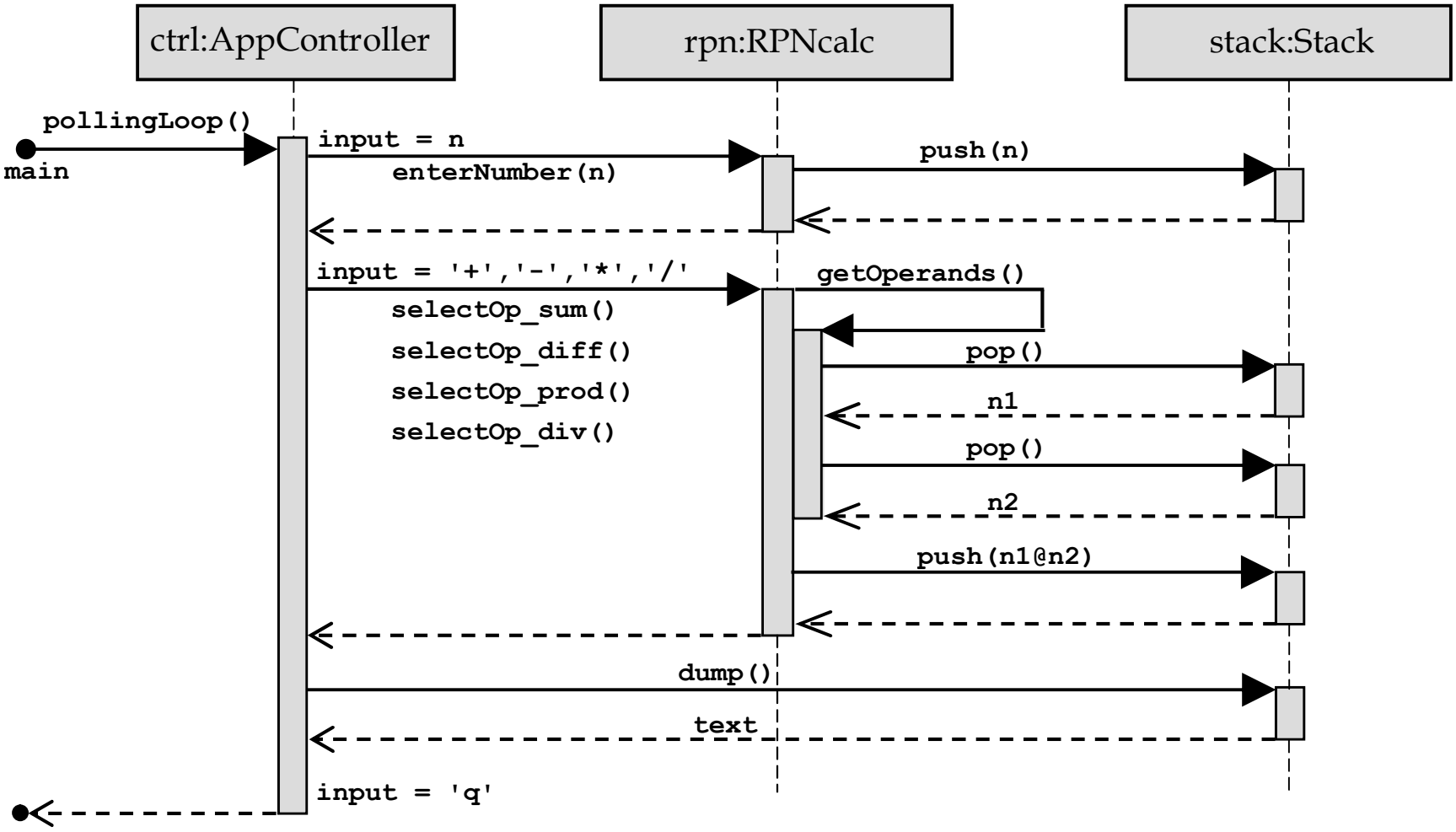
__init__: método de inicialização
self: objeto corrente

propriedade: objeto da classe RPNcalc

método pollingLoop:
loop infinito que captura um string digitado pelo usuário

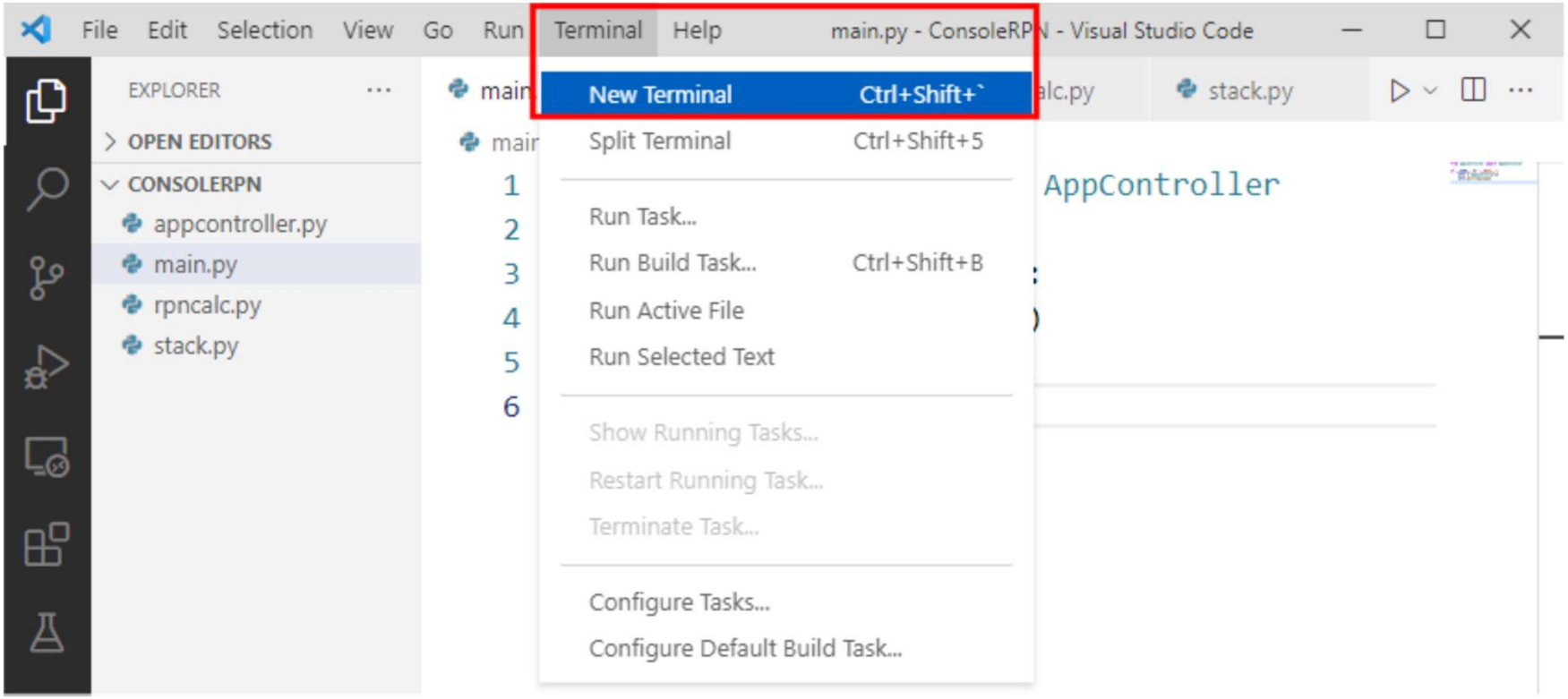
Calculadora RPN de inteiros em Python – versão console

Diagrama UML de sequência



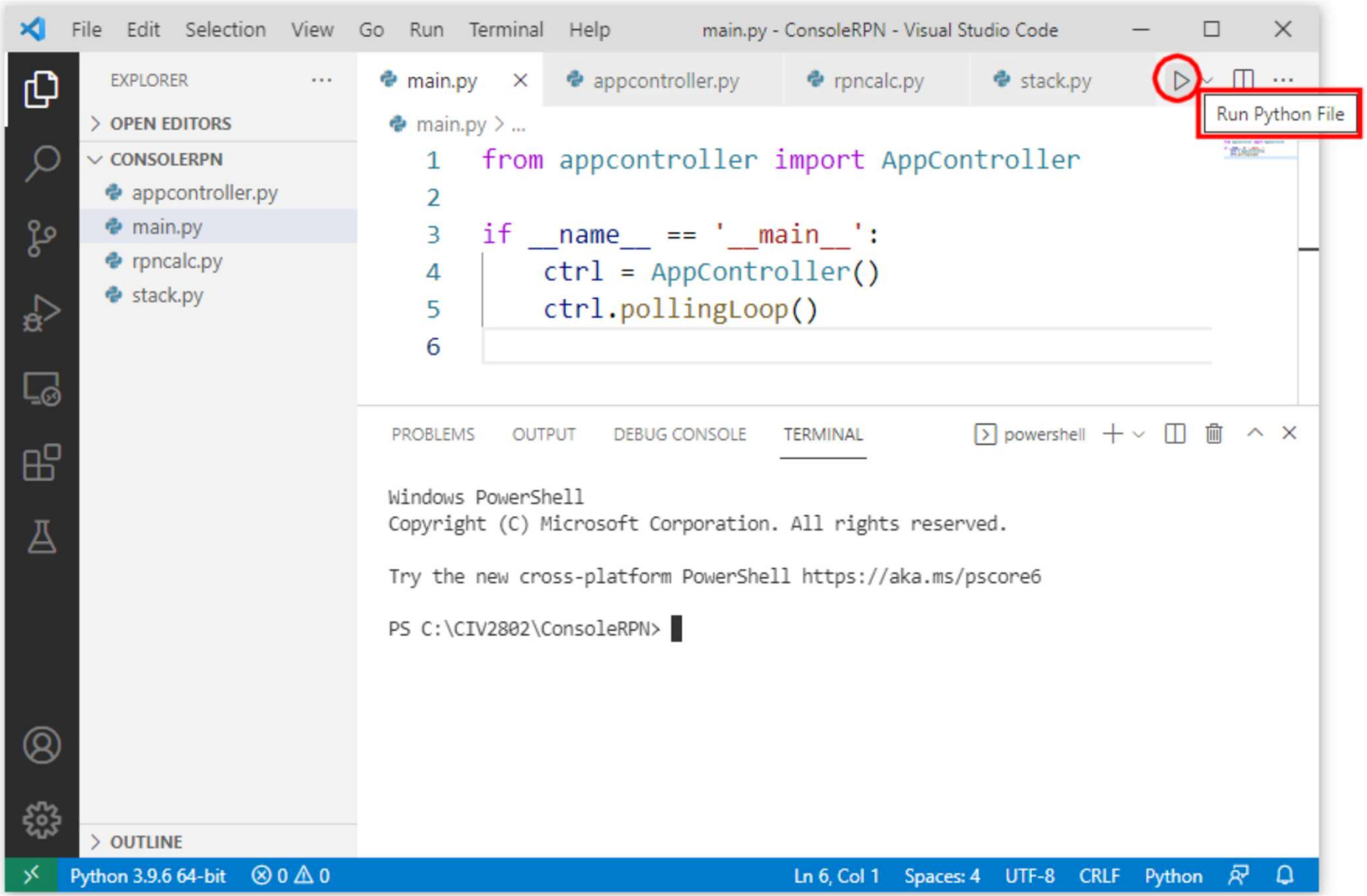
Calculadora RPN de inteiros em Python – versão console

Interface do Visual Studio Code

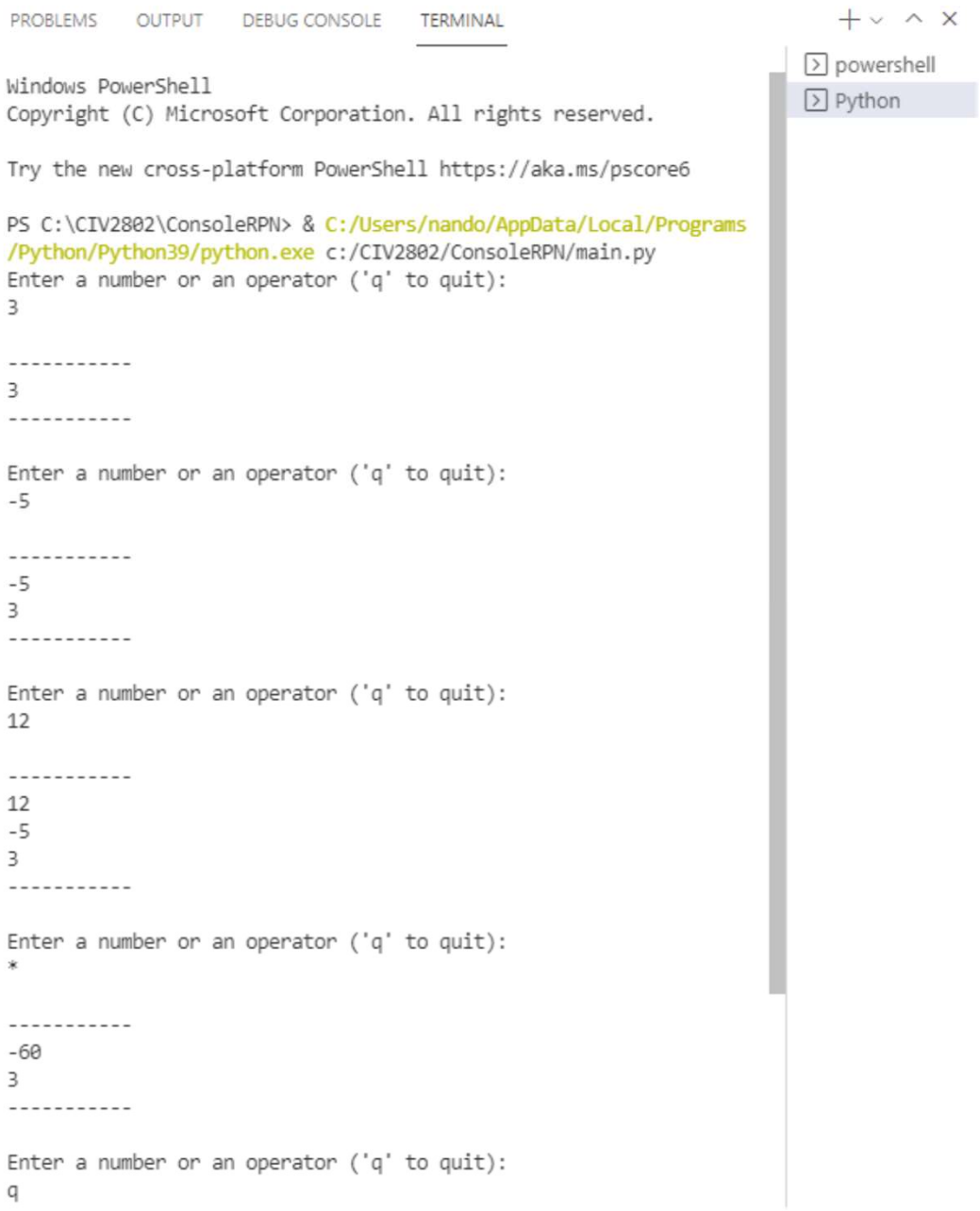


Calculadora RPN de inteiros em Python – versão console

Interface do Visual Studio Code

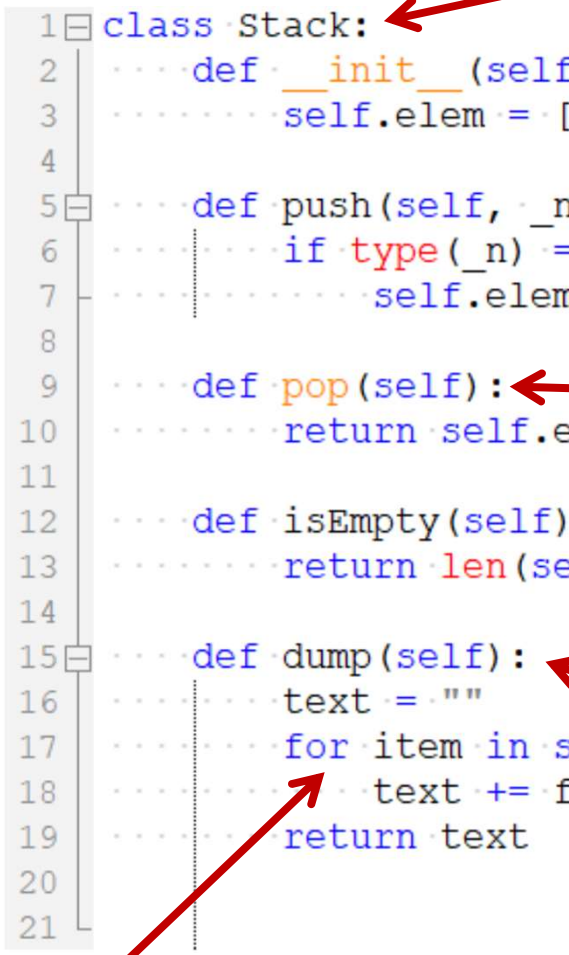


Calculadora RPN de inteiros em Python – versão console



Pilha (Stack) de inteiros em Python utilizando “list”

Arquivo stack.py:



```
class Stack:
    def __init__(self):
        self.elem = []

    def push(self, _n):
        if type(_n) == int:
            self.elem.insert(0,_n)

    def pop(self):
        return self.elem.pop(0)

    def isEmpty(self):
        return len(self.elem) == 0

    def dump(self):
        text = ""
        for item in self.elem:
            text += f"{item}\n"
        return text
```

classe Stack para pilha

__init__: método de inicialização
self: objeto corrente

propriedade elem: lista de elementos da pilha de inteiros, inicializada vazia

método push: insere um valor no início (índice 0) da lista

método pop: remove valor do início da lista e retorna esse valor

método isEmpty: retorna um booleano (true ou false) indicando se a pilha está vazia (testa se o comprimento da lista é nulo)

método dump: retorna um texto com os valores dos elementos da pilha por linha

loop for em que item assume em cada passo o valor de um elemento da lista

Orientacão a Objetos: Encapsulamento

Versão utilizando "array"

```
import numpy as np

class Stack:
    def __init__(self):
        self.top = -1
        self.elem = np.zeros(50, dtype=int)

    def push(self, _n):
        if type(_n) == int:
            self.top += 1
            self.elem[self.top] = _n

    def pop(self):
        if self.top < 0:
            return None
        n = self.elem[self.top]
        self.top -= 1
        return n

    def isEmpty(self):
        return self.top < 0

    def dump(self):
        text = ""
        for i in range(self.top,-1,-1):
            text += f"{self.elem[i]}\n"
        return text
```

Versão utilizando "list"

```
class Stack:
    def __init__(self):
        self.elem = []

    def push(self, _n):
        if type(_n) == int:
            self.elem.insert(0,_n)

    def pop(self):
        return self.elem.pop(0)

    def isEmpty(self):
        return len(self.elem) == 0

    def dump(self):
        text = ""
        for item in self.elem:
            text += f"{item}\n"
        return text
```

As estruturas de dados das duas implementações da classe Stack (utilizando "array" e utilizando "list") são encapsuladas dentro da classe.
Clientes da classe não precisam saber da diferença de implementação, e utilizam a classe através dos métodos que têm chamadas iguais nas duas versões.

Conceito de herança em programação orientada a objetos

Arquivo rpncalc.py (versão da classe RPNcalc herdando da classe Stack):

```
1  from stack import *
2
3  class RPNcalc(Stack):
4      def __init__(self):
5          super(RPNcalc, self).__init__()
6
7      def getOperands(self):
8          if self.isEmpty():
9              print("Empty stack!\n")
10             return False, 0, 0
11
12         n1 = self.pop()
13         if self.isEmpty():
14             print("Two operands needed!\n")
15             self.push(n1)
16             return False, 0, 0
17
18         n2 = self.pop()
19         return True, int(n1), int(n2)
20
21     def enterNumber(self, _n):
22         self.push(_n)
23
```

```
24     def selectOp_sum(self):
25         check, n1, n2 = self.getOperands()
26         if not check:
27             return
28         self.push(n2+n1)
29
30     def selectOp_diff(self):
31         check, n1, n2 = self.getOperands()
32         if not check:
33             return
34         self.push(n2-n1)
35
36     def selectOp_prod(self):
37         check, n1, n2 = self.getOperands()
38         if not check:
39             return
40         self.push(n2*n1)
41
42     def selectOp_div(self):
43         check, n1, n2 = self.getOperands()
44         if not check:
45             return
46         if n1 != 0:
47             self.push(int(n2/n1))
48         else:
49             print("Error: division by zero!\n")
50
```

Programação Orientada a Objetos

Conceito de Herança

- Uma classe pode ser criada a partir de outra classe (base) pelo mecanismo de **herança**.

- A classe base é chamada de **super-classe** e a classe que herda é a **sub-classe**.

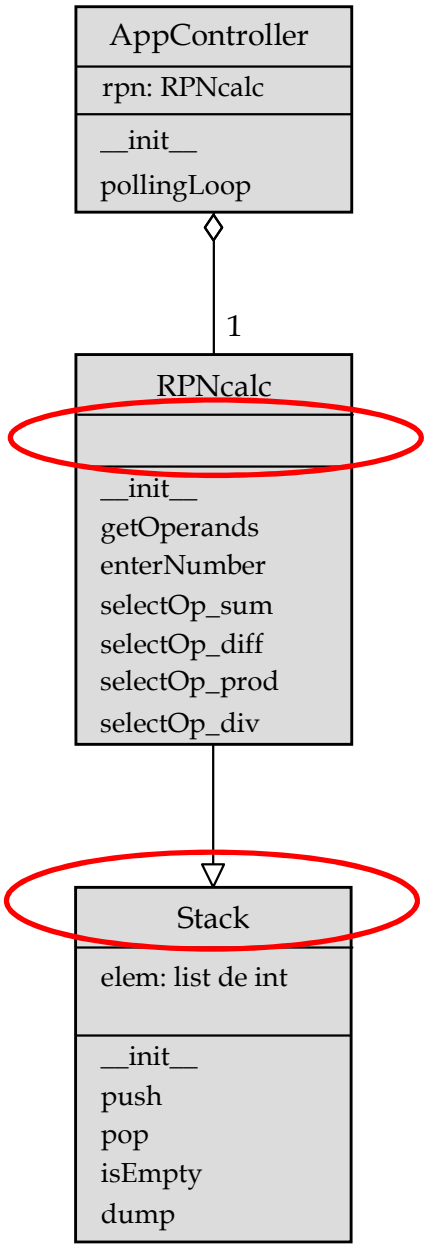
- Nesse caso, a **classe que herda é uma especialização da classe base**, pois a sub-classe é igual à super-classe com propriedades e métodos adicionais.

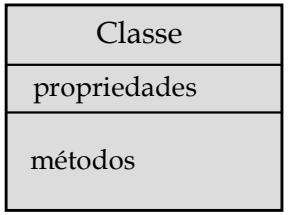
- Em um relacionamento entre classes **sem herança**, por exemplo em um **relacionamento do tipo agregação**, um objeto de uma classe **tem** uma propriedade que é um objeto da outra classe.

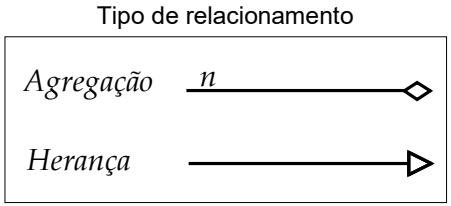
- Em um relacionamento entre classes **com herança**, um objeto da classe que herda **também é** objeto da classe base.

Diagrama UML de classes

UML: *Unified Modeling Language*







Classe AppController com classe RPNcalc que herda de Stack

Arquivo AppController.py:

```
1  from rpncalc import *
2
3  class AppController:
4      def __init__(self):
5          self.rpn = RPNcalc()
6
7      def pollingLoop(self):
8          while True:
9              print("Enter a number or an operator ('q' to quit): ")
10             string = input()
11             try:
12                 n = int(string)
13                 self.rpn.enterNumber(n)
14             except:
15                 if string == 'q':
16                     break
17                 elif string == '+':
18                     self.rpn.selectOp_sum()
19                 elif string == '-':
20                     self.rpn.selectOp_diff()
21                 elif string == '*':
22                     self.rpn.selectOp_prod()
23                 elif string == '/':
24                     self.rpn.selectOp_div()
25                 else:
26                     print("Error: enter a valid operator!\n")
27             print("\n----------\n"+self.rpn.dump()+"----------\n")
28
```