

2.Jupyter

October 27, 2019

1 Jupyter

Este notebook apresenta os seguintes tópicos:

- Section 1.1 - Sobre o Jupyter
- Section 1.1.1 - Exercício 1
- Section 1.2 - Análise de código
- Section 1.3.1 - Análise de AST
- Section 1.3.2 - Análise de complexidade ciclomática e linhas de código

1.1 Sobre o Jupyter

Ferramenta que permite combinar código, texto formatado com markdown, visualizações e interações.

O Jupyter funciona com diversas linguagens. A mais usada é Python, que é a linguagem que será usada neste minicurso.

Outras linguagens podem ser usadas a partir da instalação de Kernels diferentes.

A seguinte célula define a variável `x` e atribui o valor `1` a ela.

```
[1]: x = 1
```

A seguinte célula apresenta o valor de `x`. O Jupyter apresenta como resultado de uma célula a última expressão que aparece na mesma.

```
[2]: x
```

```
[2]: 1
```

Ao executar uma célula, o Jupyter atribui o resultado à variável `_` e adiciona no dicionário `Out`, que pode ser acessado por um índice.

Além de atribuir o resultado de uma célula, o código usado fica armazenado na lista `In`.

```
[3]: if _:  
    print("Code:", In[2])  
    print("Result:", Out[2])
```

Code: x
Result: 1

Note que a última célula apresentou os valores do print, mas não teve saída, pois não teve nenhuma expressão no final da célula. Nesse caso, o Jupyter não sobrescreve o valor de `_` e não insere nada em Out.

```
[4]: print(_)
```

1

Para evitar que uma expressão no final de uma célula seja interpretada como expressão, basta adicionar `;` a ela.

```
[5]: x + 1;
```

```
[6]: _
```

```
[6]: 1
```

1.1.1 Exercício 1

Faça um algoritmo que calcule a soma de todos elementos ímpares e soma de todos os números pares da seguinte lista e apresente como resultado da célula: [1, 51, 2, 5, 7, 0, 10, 22, 3, 4, 9, 8, 2, 6, 12, 18, 43].

```
[7]: lista = [1, 47, -2, 5, 7, 0, 12, 12, 3, 6, 9, 8, 2, 6, 2, 18, 43, 13]
    impar = 0
    par = 0
    ...
    impar, par
```

```
[7]: (128, 64)
```

1.2 Analise de código

Como temos acesso a todo o código escrito no Jupyter a partir da variável `In`, podemos usar código escrito no próprio notebook para fazer análises. A seguir contamos a quantidade de caracteres do exercício.

```
[8]: code = In[-2]
    len(code)
```

```
[8]: 179
```

Note que usamos `In[-2]` para acessar a célula anterior, pois o índice `-1` representa a própria célula.

1.3 Bibliotecas

Por usarmos Python normalmente nas células, também podemos usar as formas de import do Python, como o `import` e o `from ... import` para importar bibliotecas.

1.3.1 Análise de AST

A seguir, importaremos a biblioteca `ast` do Python para analisar o código que fizemos no exercício 1.

Se você adicionou alguma célula depois do exercício, lembrar de atualizar o índice de In.

```
[9]: import ast

class AssignmentVisitor(ast.NodeVisitor):

    def __init__(self):
        self.assignments = 0

    def visit_Assign(self, node):
        self.assignments += 1

    def visit_AugAssign(self, node):
        self.assignments += 1

tree = ast.parse(code)
visitor = AssignmentVisitor()
visitor.visit(tree)
visitor.assignments
```

[9]: 5

Nessa célula, definimos uma classe `AssignmentVisitor` que herda de `ast.NodeVisitor` e implementa as funções `visit_Assign` e `visit_AugAssign`. Essas funções são chamadas ao visitar elementos dos tipos `Assign` e `AugAssign` na árvore sintática abstrata do Python. Ao visitar esses elementos, nossa função incrementa um contador de `assignments`.

Para executar esse visitor, precisamos chamar `ast.parse` para gerar uma raiz de uma `ast` e executar `visitor.visit` para visitá-la.

A definição da AST do Python com todos os elementos possíveis pode ser encontrada na documentação oficial: <https://docs.python.org/3/library/ast.html>

Existe uma outra documentação mais completa relacionada à AST do Python (Green Tree Snakes): <https://greentreesnakes.readthedocs.io/en/latest/>

1.3.2 Complexidade ciclomatica (radon)

Além de bibliotecas builtin, também podemos importar bibliotecas externas. A seguir instalamos e usamos a biblioteca **radon**, que serve para calcular métricas do código.

Se a biblioteca já estiver instalada no ambiente, a seguinte célula não terá efeito algum. Não é necessário instalar bibliotecas sempre que for usar o notebook.

```
[10]: output = !pip install radon
      print('\n'.join(output[:5]))
```

```
Requirement already satisfied: radon in /home/joao/anaconda3/lib/python3.7/site-
packages (4.0.0)
Requirement already satisfied: flake8-polyfill in
/home/joao/anaconda3/lib/python3.7/site-packages (from radon) (1.0.2)
Requirement already satisfied: mando<0.7,>=0.6 in
/home/joao/anaconda3/lib/python3.7/site-packages (from radon) (0.6.4)
Requirement already satisfied: colorama<0.5,>=0.4 in
/home/joao/anaconda3/lib/python3.7/site-packages (from radon) (0.4.1)
Requirement already satisfied: future in
/home/joao/anaconda3/lib/python3.7/site-packages (from radon) (0.17.1)
```

Note que a célula anterior usou uma **bang expression** para executar diretamente comandos no sistema. Essas expressões fazem parte do kernel que usamos para Python (IPython).

Agora podemos importar funções que caculam a complexidade ciclomática usando **radon**.

```
[11]: from radon.complexity import cc_visit
      code_with_def = """
      def f():
          {}
      """.format(
          "\n    ".join(code.split("\n"))
      )
      cc_visit(code_with_def)
```

```
[11]: [Function(name='f', lineno=2, col_offset=0, endline=11, is_method=False,
      classname=None, closures=[], complexity=3)]
```

A função **cc_visit** apenas calcula a complexidade de funções e classes e não de código isolado. Para permitir esse cálculo, foi necessário quebrar o código em cada `\n` e inserir indentado em uma função.

A biblioteca **radon** também calcula outras métricas, como número de linhas lógicas (lloc), linhas de código (sloc), linhas de comentário (comments), linhas de comentário sem código (single_comments), strings de multilinha (multi), linhas em branco (blank), e total de linhas, respeitando a seguinte equação:

$$\$ \text{loc} = \text{sloc} + \text{blanks} + \text{multi} + \text{single_comments} \$$$

```
[12]: from radon.raw import analyze  
analyze(code)
```

```
[12]: Module(loc=9, lloc=9, sloc=9, comments=0, multi=0, blank=0, single_comments=0)
```

Continua: [3.IPython.pdf](#)