2.Jupyter

October 25, 2019

Para entrar no modo apresentação, execute a seguinte célula e pressione -

[1]:
```
%reload_ext slide
```

```
<IPython.core.display.Javascript object>
```

1 Jupyter

Este notebook apresenta os seguintes tópicos:

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.

[2]:
```
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.

[3]:
```
x
```

[3]: 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`.

[4]:
```
if _:
    print("Code:", In[3])
    print("Result:", Out[3])
```

```
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 sobreescreve o valor de `_` e não insere nada em `Out`.

[5]:
```
print(_)
```

```
1
```

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

[6]:
```
x + 1;
```

[7]:
```
_
```

[7]:
```
1
```

1.1.1 Exercicio 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]`.

[8]:
```
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
```

[8]:
```
(128, 64)
```

1.2 Analise de codigo

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.

[9]:
```
code = In[-2]
len(code)
```

[9]:
```
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 Analise 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.

[10]:
```
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
```

[10]:
```
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.

[11]:
```
!pip install radon
```

```
Collecting radon
  Using cached https://files.pythonhosted.org/packages/cf/fe/c400dbbbbde6649ad01
64ef2ffef3672baefc62ecb676f58d0f25d8f83b0/radon-4.0.0-py2.py3-none-any.whl
Requirement already satisfied: future in
/home/joao/anaconda3/lib/python3.7/site-packages (from radon) (0.17.1)
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: 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: flake8 in
/home/joao/anaconda3/lib/python3.7/site-packages (from flake8-polyfill->radon)
(3.7.8)
Requirement already satisfied: six in /home/joao/anaconda3/lib/python3.7/site-
packages (from mando<0.7,>=0.6->radon) (1.12.0)
Requirement already satisfied: pycodestyle<2.6.0,>=2.5.0 in
/home/joao/anaconda3/lib/python3.7/site-packages (from
flake8->flake8-polyfill->radon) (2.5.0)
Requirement already satisfied: pyflakes<2.2.0,>=2.1.0 in
/home/joao/anaconda3/lib/python3.7/site-packages (from
flake8->flake8-polyfill->radon) (2.1.1)
Requirement already satisfied: entrypoints<0.4.0,>=0.3.0 in
/home/joao/anaconda3/lib/python3.7/site-packages (from
flake8->flake8-polyfill->radon) (0.3)
Requirement already satisfied: mccabe<0.7.0,>=0.6.0 in
/home/joao/anaconda3/lib/python3.7/site-packages (from
flake8->flake8-polyfill->radon) (0.6.1)
Installing collected packages: radon
Successfully installed radon-4.0.0
```

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`.

[12]:
```
from radon.complexity import cc_visit
code_with_def = """
def f():
    {}
""".format(
    "\n    ".join(code.split("\n"))
)

cc_visit(code_with_def)
```

[12]:
```
[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:

$ loc = sloc + blanks + multi + single_comments $

[13]:
```
from radon.raw import analyze
analyze(code)
```

[13]:
```
Module(loc=9, lloc=9, sloc=9, comments=0, multi=0, blank=0, single_comments=0)
```

Continua: 3.IPython.pdf