3.IPython

October 25, 2019

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

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

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

1 IPython

Este notebook apresenta os seguintes tópicos:

1.1 Magicas do IPython

Na parte anterior do minicurso, apresentamos **bang expression** como uma extensão da linguagem Python fornecida pelo kernel IPython para executar comandos no sistema.

Além dessa extensão, o IPython também permite escrever "mágicas"/"magics" que modificam a forma de executar operações. Existem duas principais formas de "magics":

- line magic: altera o restante da linha
- cell magic: altera a célula inteira

1.1.1 Line magic

A seguir temos um exemplo de line magic que mostra o histórico de células executadas com a numeração de células.

[2]:
```
a = 1
```

[3]:
```
b = a
```

[4]:
```
%history -n
```

```
  1: %reload_ext slide
  2: a = 1
  3: b = a
  4: %history -n
```

Essa line magic apenas imprimiu o histórico, porém existem outras que podem ser usadas no meio de expressões do Python, como a `%who_ls`, que retorna todas as variáveis de um determinado tipo definidas no notebook.

[5]:
```
variaveis = %who_ls int
for var, _ in zip(variaveis, range(5)):
    print(var, eval(var))
```

```
a 1
b 1
```

Além de estender a sintaxe do Python para adicionar bang expressions e magics, o IPython também permite consultar a documentação de módulos, classes, funções e magics, ao adicionar `?` após o nome.

[6]:
```
%who_ls?
```

O uso de duas interrogações (`??`) exibe o código fonte.

[7]:
```
%who_ls??
```

1.1.2 Cell magic

Cell magics permitem alterar a execução de uma célula por completo. A cell magic a seguir executa código javascript no navegador.

[8]:
```
%%javascript

console.log("Teste")
```

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

Já a cell magic a seguir calcula o tempo de execução de uma célula Python.

[9]:
```
%%time
from time import sleep
sleep(2)
```

```
CPU times: user 702 µs, sys: 93 µs, total: 795 µs
Wall time: 2 s
```

1.1.3 Como ocorre a execução

A line magic `%history` apresentada anteriormente pode ser usada para entender o que o IPython está fazendo quando usamos essas magics. Para isso, precisamos ver o histórico traduzido para Python, utilizando a flag `-t`.

[10]:
```
%history -t -l 6
```

```
get_ipython().run_line_magic('history', '-n')
variaveis = get_ipython().run_line_magic('who_ls', 'int')
for var, _ in zip(variaveis, range(5)):
    print(var, eval(var))
get_ipython().run_line_magic('pinfo', '%who_ls')
get_ipython().run_line_magic('pinfo2', '%who_ls')
get_ipython().run_cell_magic('javascript', '', '\nconsole.log("Teste")\n')
get_ipython().run_cell_magic('time', '', 'from time import sleep\nsleep(2)\n')
```

Note os seguintes comandos:

```
get_ipython().run_cell_magic('time', '', 'from time import sleep\nsleep(2)\n')
get_ipython().run_line_magic('who_ls', 'int')
```

Eles indicam o que o shell do IPython (resultado de `get_ipython()`) deve executar. A função indica se deve executar cell magic ou line magic. O primeiro parâmetro indica o nome da magic. Por fim, os últimos parâmetros indicam os parâmetros para a função da magic.

Esses comandos podem ser executados diretamente no notebook:

[11]:
```
get_ipython().run_line_magic('who_ls', 'int')
```

[11]:
```
['a', 'b']
```

1.1.4 Lista de magics

Podemos usar a magic `%lsmagic` para listar quais são todas as magics do IPython e a magic `%magic` para entender como funciona a parte de magics.

[12]:
```
%lsmagic
```

[12]:
```
Available line magics:
%alias  %alias_magic  %autoawait  %autocall  %automagic  %autosave  %bookmark
%cat  %cd  %clear  %colors  %conda  %config  %connect_info  %cp  %debug  %dhist
%dirs  %doctest_mode  %ed  %edit  %env  %gui  %hist  %history  %killbgscripts
%ldir  %less  %lf  %lk  %ll  %load  %load_ext  %loadpy  %logoff  %logon
%logstart  %logstate  %logstop  %ls  %lsmagic  %lx  %macro  %magic  %man
%matplotlib  %mkdir  %more  %mv  %notebook  %page  %pastebin  %pdb  %pdef  %pdoc
%pfile  %pinfo  %pinfo2  %pip  %popd  %pprint  %precision  %prun  %psearch
%psource  %pushd  %pwd  %pycat  %pylab  %qtconsole  %quickref  %recall  %rehashx
%reload_ext  %rep  %rerun  %reset  %reset_selective  %rm  %rmdir  %run  %save
```

```
%sc  %set_env  %store  %sx  %system  %tb  %time  %timeit  %unalias  %unload_ext
%who  %who_ls  %whos  %xdel  %xmode

Available cell magics:
%%!  %%HTML  %%SVG  %%bash  %%capture  %%debug  %%file  %%html  %%javascript
%%js  %%latex  %%markdown  %%perl  %%prun  %%pypy  %%python  %%python2
%%python3  %%ruby  %%script  %%sh  %%svg  %%sx  %%system  %%time  %%timeit
%%writefile

Automagic is ON, % prefix IS NOT needed for line magics.
```

Perceba que automagic está ativo, isso significa que podemos usar line magics sem % explícito:

[13]:
```
who_ls int
```

[13]:
```
['a', 'b']
```

Para outras magics, veja o arquivo InteratividadeExtra.ipynb

1.1.5 Registrando novas magics

Agora que sabemos como o IPython executa as magics, podemos pensar em criar e registrar novas magics.

[14]:
```
from IPython.core.magic import Magics, magics_class, cell_magic

@magics_class
class LenMagic(Magics):
    @cell_magic
    def size(self, line, cell):
        return len(cell)
```

Em seguida registramos a magic:

[15]:
```
shell = get_ipython()
shell.register_magics(LenMagic)
```

Com isso, podemos usar para obter o tamanho de códigos de células:

[16]:
```
%%size
print("a")
```

[16]:
```
11
```

Note que o conteúdo da célula não foi executado. Ao invés disso, ele foi passado para a função size que o processou e retornou 11

Agora vamos para um exemplo mais complicado, com argumentos, criação dinâmica de classes e análise da AST.

[17]:
```
import ast
from IPython.core.magic_arguments import magic_arguments, argument,␣
 ↪parse_argstring

@magics_class
class ASTMagic(Magics):

    @magic_arguments()
    @argument(
        "methods",
        default=["visit_Assign", "visit_AugAssign"],
        nargs="*",
        help="method names to be defined on AST Visitor"
    )
    @cell_magic
    def count_ast(self, line, cell):
        args = parse_argstring(self.count_ast, line)
        class CustomVisitor(ast.NodeVisitor):
            def __init__(self):
                self.count = 0

            def _increment_counter(self, node):
                self.count += 1

        for method in args.methods:
            setattr(CustomVisitor, method, CustomVisitor._increment_counter)

        tree = ast.parse(cell)
        visitor = CustomVisitor()
        visitor.visit(tree)
        return visitor.count

shell = get_ipython()
shell.register_magics(ASTMagic)
```

Neste exemplo, definimos argumentos usando decoradores e usamos a função `parse_argstring` para transformá-los em uma estrutura. A definição segue o `argparse` do Python: https://docs.python.org/3/library/argparse.html

```
@magic_arguments()
@argument(
    "methods",
    default=["visit_Assign", "visit_AugAssign"],
    nargs="*",
    help="method names to be defined on AST Visitor"
```

)

Além da parte dos argumentos, criamos classes dinamicamente dentro da função e definimos os métodos dela como sendo referências ao método `_increment_counter`.

```
for method in args.methods:
    setattr(CustomVisitor, method, CustomVisitor._increment_counter)
```

Por fim, executamos o visitor e retornamos a contagem.

```
tree = ast.parse(cell)
visitor = CustomVisitor()
visitor.visit(tree)
return visitor.count
```

[18]:
```
%%count_ast
def f():
    pass
a = 1
b = 2
c = 3
```

[18]:
```
3
```

[19]:
```
%%count_ast visit_FunctionDef
def f():
    pass
a = 10
b = 2
c = 3
```

[19]:
```
1
```

1.2 Exercicio 2

Modifique a magic `count_ast` para retornar um dicionário ou counter com uma contagem de todos os nós da ast. O nome da magic resultante deve ser `ast_counter`.

Dicas: - Use o método `generic_visit(self, node)` para visitar os nós da AST sem especificar o nome - Obtenha o nome do elemento na AST usando `type(node).__name__` - Visite nós recursivamente

[20]:
```
...
```

[21]:
```
%%ast_counter
def f():
    pass
a = 1
b = 2
c = 3
```

[21]:
```
Counter({'Module': 1,
         'FunctionDef': 1,
         'arguments': 1,
         'Pass': 1,
         'Assign': 3,
         'Name': 3,
         'Store': 3,
         'Num': 3})
```

1.3 Exercicio 3

Crie uma magic, `%%radon`, que utilize radon para extrair informações de complexidade ciclomática e linhas de código de uma célula.

[22]:
```
from radon.raw import analyze
from radon.complexity import cc_visit

template = """
def __radon_analysis():
    {}
"""

...
```

[23]:
```
%%radon
def f():
    pass
a = 1
if a:
    b = 2
    if b:
        c = 3
```

[23]:
```
(Module(loc=7, lloc=7, sloc=7, comments=0, multi=0, blank=0, single_comments=0),
 [Function(name='__radon_analysis', lineno=2, col_offset=0, endline=9,
is_method=False, classname=None, closures=[Function(name='f', lineno=3,
col_offset=4, endline=4, is_method=False, classname=None, closures=[],
complexity=1)], complexity=3)])
```

1.4 Exercicio 4

Faça uma line magic para clonar repositórios do GitHub recebendo o repositório no formato `Organizacao/Repositorio` e com argumentos para especificar o diretório e o commit.

Exemplo de uso:

```
%clone gems-uff/sapos -d repos/sapos -c a9b0f7b3
```

Dicas:

- Você pode usar **bang expressions** para chamar os comandos `git clone` e `git checkout`.
- Bang expressions aceitam combinar variáveis do Python usando usando `{variavel}`, entre chaves
- A URL de um repositório do tipo `owner/name` no GitHub é `https://github.com/owner/name.git`

[24]:
```
from IPython.core.magic import line_magic

...
```

[25]:
```
%clone gems-uff/sapos -d repos/sapos -c a9b0f7b3
```

```
Cloning into 'repos/sapos'…
remote: Enumerating objects: 22, done.
remote: Counting objects: 100% (22/22), done.
remote: Compressing objects: 100% (19/19), done.
remote: Total 12954 (delta 4), reused 11 (delta 3), pack-reused 12932
Receiving objects: 100% (12954/12954), 10.41 MiB | 8.38 MiB/s, done.
Resolving deltas: 100% (8011/8011), done.
/home/joao/projects/tutorial/repos/sapos
Note: checking out 'a9b0f7b3'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at a9b0f7b Merge branch 'master' into bugfixes
/home/joao/projects/tutorial
```

Continua: 4.Proxy.pdf