pythondrops.com

Curso Básico de Python

Lição 4: Organizando o código

Sumário

Organização

Há muitas maneiras de organizar um código-fonte, dividindo-o em módulos. Temos: funções, OOP, programação funcional e pacotes. Vamos ver como fazer isso em Python.

Zen of Python

A proposta PEP 20 estabelece algumas diretrizes que devem orientar os desenvolvedores Python:

https://www.python.org/dev/peps/pep-0020/

Se você quiser, inicie o Python interativo e digite: “import this”.

```
Python 3.7.0 (default, Jun 28 2018, 13:15:42)
[GCC 7.2.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Você deve ler essas diretrizes e procurar aplicá-las no seu dia a dia. Ao longo do curso, vou mencionar e explicar com exemplos algumas delas.

PEP 8

Há uma especificação (PEP = Python Enhancement Proposal) que é um guia de estilo de programação em Python. Há muita coisa lá e eu recomendo que você leia com calma:

https://www.python.org/dev/peps/pep-0008/

Porém, o mais importante neste momento do seu aprendizado é entender o estilo de nomenclatura mais aceito em Python:

- Nomes de variáveis e funções: Em minúsculas, com palavras separadas por caractere sublinha (“_”). Exemplos: “codigo_cliente”, “calcular_divida()”;
- Nomes de constantes (*): Em maiúsculas, com palavras separadas por “_”. Exemplos: “PI”, “INDICE”, “FATOR_REDUCAO”;
- Classes (veremos em outra lição): Nomes de classes seguem o padrão “corcova de camelo”, com a inicial maiúscula e as palavras separadas por letras maiúsculas. Exemplos: “ItemCompra”, “ClienteEspecial”;
- Métodos e variáveis de instância (veremos em outra lição): Seguem o padrão de nomes de variáveis e funções comuns;
- Nomes de módulos: Seguem o padrão de nomes de variáveis e funções;

(*) Em Python não temos constantes de verdade. Cabe a você não alterar o seu valor.

Por exemplo, em C++:

```
const int FATOR = 5;
#define CODIGO_ESPECIAL 10
```

Em Java:

private static final int FATOR = 5;

Em Python:

```
FATOR = 5
def CODIGO_ESPECIAL():
      return 10
```

Em Python, mesmo que você declare uma variável de escopo de módulo, com seu nome todo em maiúsculas, ela ainda pode ter seu valor alterado. Uma alternativa é declarar uma função e usá-la como constante.

Funções

Como toda linguagem de programação moderna, Python permite modularizar seu código-fonte através da criação de funções. São subrotinas que podem ou não retornar valores e podem ou não receber parâmetros. Declaramos funções com “def”:

```
>>> def PI():
...     return 22/7
...
>>> print(PI())
3.142857142857143
```

Todo o bloco de comandos que faz parte da função deve ser identado. O padrão é identar com quatro espaços. Note que ao final da declaração da função, deve haver um “:”.

Tenha ou não parâmetros (ou argumentos), é preciso informar os parêntesis na declaração e na invocação da função.

Retorno

Uma função pode ou não retornar um valor e podemos ou não ignorá-lo:

```
>>> def calcular(valor,taxa,prazo):
...     return valor + valor*(taxa/100)*prazo
...
>>> montante = calcular(1000.00,2.5,10)
>>> print(montante)
1250.0
```

Uma função pode retornar mais de um valor:

```
>>> def calcular(valor,taxa,prazo):
...     juros = valor * (taxa/100) * prazo
...     montante = valor + juros
...     return juros,montante
...
>>> j,m = calcular(1000.00,2.5,10)
>>> print('Juros: {}, Montante: {}'.format(j,m))
Juros: 250.0, Montante: 1250.0
```

Parâmetros

Os parâmetros podem ser referenciados de maneira posicional ou através de seu nome:

```
>>> j,m = calcular(1000.00,prazo=10,taxa=2.5)
>>> print('Juros: {}, Montante: {}'.format(j,m))
Juros: 250.0, Montante: 1250.0
```

Neste exemplo, invocamos a mesma função “calcular” passando o valor como parâmetro posicional (ele realmente fica na primeira posição, na ordem da função), mas invertemos a ordem dos parâmetros “taxa” e “prazo”, para isto, informamos seus nomes. Você pode informar parâmetros de maneira posicional ou não.

Também podemos criar parâmetros opcionais, informando um valor “default”:

```
>>> def calcular(valor,taxa,prazo=1):
...     juros = valor * (taxa/100) * prazo
...     montante = valor + juros
...     return juros,montante
...
>>> j,m = calcular(1000.00,2.5)
>>> print('Juros: {}, Montante: {}'.format(j,m))
Juros: 25.0, Montante: 1025.0
```

Ao atribuir um valor na declaração do parâmetro, você o torna opcional. Se um argumento é opcional, os seguintes a ele também devem ser. Por exemplo, vamos supor que a taxa tenha um valor default, mas o prazo não:

```
>>> def calcular(valor,taxa=2.5,prazo):
...     juros = valor * (taxa/100) * prazo
...     montante = valor + juros
...     return juros,montante
...
  File "<stdin>", line 1
SyntaxError: non-default argument follows default argument
```

Ele está reclamando que há um argumento (ou parâmetro) sem valor default, depois de um argumento que possui valor default. Argumentos com valor default são opcionais. Porém, você pode ter todos os parâmetros como opcionais:

```
>>> def calcular(valor=10,taxa=2.5,prazo=1):
...     juros = valor * (taxa/100) * prazo
...     montante = valor + juros
...     return juros,montante
...
>>> print(calcular())
(0.25, 10.25)
```

Vamos supor que você queira informar apenas a taxa, deixando os outros dois argumentos com os valores default. Para isto, tem que usar a notação com nomes de parâmetros:

```
>>> print(calcular(taxa=1.2))
(0.12, 10.12)
```

Parâmetros variáveis

Podemos ter uma quantidade indeterminada de parâmetros em uma função:

```
>>> def media(*parcelas):
...     acumulado = 0.0
...     for p in parcelas:
...         acumulado = acumulado + p
...     return acumulado / len(parcelas)
...
>>> print(media(10.0,7.8,5.0,7.5))
7.575
```

Usamos a notação “*” antes do nome da variável que receberá a lista de parâmetros.

E podemos ter uma lista de parâmetros do tipo chave/valor (kwargs):

```
>>> def funcao(**elementos):
...     for chave,valor in elementos.items():
...         print('Chave: {}, valor: {}'.format(chave,valor))
...
>>> funcao(nome='fulano',codigo=5,especial=True)
Chave: nome, valor: fulano
Chave: codigo, valor: 5
Chave: especial, valor: True
```

Quando utilizamos “**” na frente do parâmetro, na declaração da função, ele entende que os parâmetros serão passados como um dicionário Python.

Módulos

Um módulo Python é uma unidade de código, ou um arquivo-fonte, que expõe declarações para serem utilizadas por outros scripts. Entenda como uma “biblioteca” de código Python.

Vamos imaginar que você crie um conjunto de funções e queira utilizá-las em outros programas. Por exemplo, vamos imaginar que você tenha um jogo e precise criar um tabuleiro virtual bidimensional. Você quer mover objetos (retângulos) e saber se colidiram, mas tem que tomar cuidado com as bordas.

O código “geom.py” faz isto. Veja no repositório, dentro da mesma pasta “licao4”. Nele, há duas funções:

```
def colisao(objeto1,objeto2):
    if objeto1['x1']> objeto2['x2']or objeto2['x1']> objeto1['x2']:
        return False
    if objeto1['y1'] < objeto2['y2'] or objeto2['y1'] < objeto1['y2']:
        return False
    return True

def mover(objeto,direcao):
    novo=GABARITO_OBJETO.copy()
    novo['x1']=objeto['x1']+DIRECAO[direcao][0]
    novo['y1']=objeto['y1']+DIRECAO[direcao][1]
    novo['x2']=objeto['x2']+DIRECAO[direcao][0]
    novo['y2']=objeto['y2']+DIRECAO[direcao][1]
    if novo['x1']<0 or novo['x2']==LARGURA:
        return False
    if novo['y1']==ALTURA or novo['y2']<0:
        return False
    objeto['x1']=novo['x1']
    objeto['y1']=novo['y1']
    objeto['x2']=novo['x2']
    objeto['y2']=novo['y2']
    return True
```

E temos um código imediato que serve para testar as funções. Agora, vamos tentar usar essa biblioteca dentro do nosso script. Crie um script assim:

```
import geom
#
# Game loop
#
# Criar objeto:
player=geom.GABARITO_OBJETO.copy()
```

```
# Mover objeto:
retorno = geom.mover(player,'cima')
if not retorno:
    print('erro')
```

Perfeito, não? Importamos todas as declarações de “geom” em “jogo”, e podemos usar o prefixo “geom” como namespace. Agora, tente executar o jogo:

```
python jogo.py
True
False
Moveu objeto1 para a direita:  {'x1': 1, 'y1': 10, 'x2': 11, 'y2': 0}
Moveu objeto3 para cima:  {'x1': 15, 'y1': 16, 'x2': 20, 'y2': 11}
Moveu objeto3 para esquerda:  {'x1': 14, 'y1': 16, 'x2': 19, 'y2': 11}
Moveu objeto3 para baixo:  {'x1': 14, 'y1': 15, 'x2': 19, 'y2': 10}
Não pode mover para a direita:  {'x1': 989, 'y1': 999, 'x2': 999, 'y2': 989}
Não pode mover para cima:  {'x1': 989, 'y1': 999, 'x2': 999, 'y2': 989}
Não pode mover para a esquerda:  {'x1': 0, 'y1': 10, 'x2': 10, 'y2': 0}
Não pode mover para baixo:  {'x1': 0, 'y1': 10, 'x2': 10, 'y2': 0}
```

Ué?! Por que ele mostrou essas linhas? Quando importamos um módulo, o seu código imediato é executado. Serve como um “startup”, caso ele seja invocado diretamente pelo Python. Para evitarmos isto, usamos uma variável ***dunder*** especial chamada: “**__name__**”. Note que dentro de “geom.py” eu inseri o comando:

```
print(__name__)
```

Quando executamos diretamente o script “geom.py”, ele mostra: “__main__”, que é o módulo principal da aplicação, criado diretamente pelo interpretador Python. Porém, quando executamos “jogo.py”, ele mostra “geom”, que é o nome do módulo “geom.py”, importado a partir do “jogo.py”.

Moral da história: O módulo que foi executado diretamente pelo Python passa a ser o “__main__”, o que podemos testar na variável “__name__”. Então, podemos colocar um “if” dentro de “geom.py”, de modo a só executarmos o código imediato se ele for o módulo “__main__”:

```
if __name__=='__main__':
    print(__name__)
    objeto1=GABARITO_OBJETO.copy()
    objeto2=GABARITO_OBJETO.copy()
    objeto3=GABARITO_OBJETO.copy()
…
```

O arquivo “geom_lib.py” já está assim. E modificamos também o arquivo “jogo.py”, criando o “jogo_novo.py”:

```
import geom_lib as geom
#
# Game loop
#
# Criar objeto:
player=geom.GABARITO_OBJETO.copy()

# Mover objeto:
retorno = geom.mover(player,'cima')
if not retorno:
    print('erro')
```

Usei uma forma diferente do ***import*** que permite renomear o namespace. Assim, não tive que alterar o resto do código: “import biblioteca as nome”.

Pasta como módulo

Seu módulo não precisa ficar na mesma pasta. Na verdade, você pode criar pastas diferentes, dependendo do tipo de módulo que vai importar. Por exemplo, veja a pasta “utils”, que contém uma versão do “geom_lib.py” e veja o script “jogo_pasta.py” que a importa:

```
import utils.geom_lib as geom
#
# Game loop
#
# Criar objeto:
player=geom.GABARITO_OBJETO.copy()

# Mover objeto:
retorno = geom.mover(player,'cima')
if not retorno:
    print('erro')
```

Agora, temos um módulo “utils” e um submódulo “geom_lib”. Para não mudar o código do jogo, eu usei a sintaxe “import as...”.

Note que dentro da pasta “utils” há um script “__init__.py”, que serve para prevenir conflito entre os nomes de pastas e os pacotes já existentes. Mas também serve para executar código de inicialização de um pacote. Geralmente, “__init__.py” fica vazio, mas eu coloquei um ***print()*** só para mostrar.

Conteúdo de um módulo

A função dir() exibe todos os elementos de um módulo importado, que estão disponíveis para uso. Por exemplo, dentro do Python interativo importei o módulo e rodei o comando:

```
>>> import utils.geom_lib
Rodou init
>>> dir(utils.geom_lib)
['ALTURA', 'DIRECAO', 'GABARITO_OBJETO', 'LARGURA', '__builtins__',
'__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__',
'__spec__', 'colisao', 'mover', 'objetos']
```

Note que importei sem a opção “import as...”, portanto, tenho que me referir ao módulo com seu nome completo: utils.geom_lib. Alguns elementos eu mesmo criei, outros, o Python criou.

Há uma terceira forma de importar elementos de um módulo, que nos permite filtrar o que queremos importar: “from <módulo> import elemento [,outro elemento, etc]”. Veja um exemplo:

```
>>> from utils.geom_lib import mover
Rodou init
>>> dir(mover)
['__annotations__', '__call__', '__class__', '__closure__', '__code__',
'__defaults__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__get__', '__getattribute__', '__globals__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__kwdefaults__', '__le__',
'__lt__', '__module__', '__name__', '__ne__', '__new__', '__qualname__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__']
>>> dir(utils.geom_lib)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'utils' is not defined
>>> from utils.geom_lib import mover,colisao
>>> dir(utils.geom_lib)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'utils' is not defined
```

Neste exemplo começamos importando só a função ***mover()***. Note que nada mais de ***utils.geom_lib*** está disponível. Depois, importamos as funções: ***mover()*** e ***colisao()***.

Podemos importar todos os elementos de um módulo de duas maneiras:

- import <módulo>
- from <módulo> import *

A primeira maneira importa tudo e coloca os elementos dentro do namespace do módulo (a não ser que você use a opção “as <nome>”). A segunda, importa também todos os elementos, mas coloca no namespace do módulo principal. Veja só:

```
>>> from utils.geom_lib import *
Rodou init
>>> dir(utils.geom_lib)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'utils' is not defined
>>> dir()
['ALTURA', 'DIRECAO', 'GABARITO_OBJETO', 'LARGURA', '__annotations__',
'__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__',
'colisao', 'mover', 'objetos']
```

O namespace utils.geom_lib nem existe no módulo principal, e todos os seus elementos estão diretamente ligados ao módulo principal (***dir()*** mostra todos os elementos do módulo principal). Podemos utilizá-los diretamente. Agora, da outra maneira é diferente:

```
>>> import utils.geom_lib
Rodou init
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'utils']
>>> dir(utils)
['__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__path__', '__spec__', 'geom_lib']
>>> dir(utils.geom_lib)
['ALTURA', 'DIRECAO', 'GABARITO_OBJETO', 'LARGURA', '__builtins__',
'__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__',
'__spec__', 'colisao', 'mover', 'objetos']
```

Agora, os elementos não estão no módulo principal, exceto o módulo ***utils***. E, ao listarmos ***utils***, encontramos ***geom_lib***. Todos os seus elementos podem ser acessados a partir do namespace ***utils.geom_lib***.

Se você examinar o código de utils.geom_lib, notará uma função cujo nome inicia por sublinha:

```
def _privfunc():
    pass
```

Qualquer elemento cujo nome inicie por sublinha é considerado privado e não é importado. Note também o comando “pass”, que não executa operação alguma. É só para guardar espaço.

Podemos controlar o que será importado utilizando a variável “__all__” declarada dentro do “__init__.py” de um módulo:

```
__all__=['mover','colisao']
```

Se quisermos permitir a importação de elementos privados, basta acrecentar à lista da variável “__all__”.

Escopo de variáveis

Em Python temos o escopo global (do módulo) e o local (de função):

```
>>> glob='Variável global'
>>>
>>> def funcao():
...     local='Variável local'
...     print(glob,local)
...
>>> funcao()
Variável global Variável local
```

Mas, ao contrário de outras linguagens, como Java, não existe o escopo de bloco de comandos. Veja este exemplo:

```
>>> glob='Variável global'
>>>
>>> def funcao():
...     local='Variável local'
...     while True:
...         var_bloco='Var bloco'
...         break
...     print(glob,local,var_bloco)
...
>>> funcao()
Variável global Variável local Var bloco
```

Eu criei uma variável chamada ***var_bloco*** dentro do ***while***, mas ela continuou a existir depois de sair do bloco de código.

Alteração de variáveis globais

Python tem suas idiossincrasias. Uma delas é a questão das variáveis globais. Me diga o que faz este código:

```
>>> achou=False
>>> def procurar(texto):
...     for c in texto:
...         if c=='*' or c=='&':
...             achou=True
...
>>> procurar('Este * um t&xto legal')
>>> print(achou)
False
```

Macacos me mordam! Ele deveria encontrar os caracteres dentro do texto, não? O que aconteceu?

Na verdade, criamos uma variável global "achou" (eu sei, essa é uma decisão ruim, mas é para exemplificar). E a tornamos verdadeira quando a função encontra algum dos caracteres. Só que as variáveis globais só ficam disponíveis para leitura. Ao tentar alterar uma variável global dentro de uma função, o Python cria uma variável local com o mesmo nome, evitando alterar a variável global. É um mecanismo de defesa. Se quisermos alterar a variável global dentro da função, demos que usar a declaração ***global***:

```
>>> achou=False
>>> def procurar(texto):
...     global achou
...     for c in texto:
...         if c=='*' or c=='&':
...             achou=True
...
>>> procurar('Este * um t&xto legal')
>>> print(achou)
True
```

De qualquer forma, essa é uma prática ruim (alterar variáveis globais dentro de funções). Nossas funções devem ser "puras", ou seja, seu resultado deveria ser determinado apenas pelos parâmetros que recebe e não deveriam alterar nada que esteja fora de seu escopo:

https://pt.stackoverflow.com/questions/255557/o-que-%C3%A9-uma-fun%C3%A7%C3%A3o-pura

Escopo local superior

O Python 3 incluiu a declaração ***nonlocal*** para permitir alteração de variáveis locais de escopo superior. Vou demonstrar isso com um exemplo do Stackoverflow (https://stackoverflow.com/questions/1261875/python-nonlocal-statement):

```
x = 0
def outer():
    x = 1
    def inner():
        x = 2
        print("inner:", x)

    inner()
    print("outer:", x)

outer()
print("global:", x)

# inner: 2
# outer: 1
# global: 0
```

Nem o “x” global nem o “x” do escopo da função “outer()” foram alterados. Para permitir que a função “inner()” altere o valor de “x” no escopo da função “outer()”, precisamos incluir a declaração ***nonlocal***:

```
x = 0
def outer():
    x = 1
    def inner():
        nonlocal x
        x = 2
        print("inner:", x)

    inner()
    print("outer:", x)

outer()
print("global:", x)

# inner: 2
# outer: 2
# global: 0
```

Exercício

Vamos exercitar os músculos!

Crie uma biblioteca contendo uma função para encontrar números primos, usando o “Crivo” de Eratóstenes, utilizando-a em um progama que fatora um número, imprimindo seus fatores primos.

A biblioteca deve estar em uma subpasta.

Você pode ler o número a ser fatorado diretamente do argumento da linha de comando:

```
import sys
…
numero=int(sys.argv[1])
```

O primeiro argumento é sempre o nome do script.

Ah, e você vai precisar encontrar uma lista de números primos, de modo a fatorar o número dado. Para isto, utilize o Crivo de Eratóstenes. Há um algoritmo na Wikipedia:

https://pt.wikipedia.org/wiki/Crivo_de_Erat%C3%B3stenes

A correção está na pasta “correção”, mas você deve tentar fazer o exercício.