Uma breve revisão de Python

Este bloco de notas tem como objetivo revisar conceitos básicos de programação em Python (versão 3) utilizando o Jupyter Notebook. Assumimos uma familiaridade mínima com a linguagem, geralmente adquirida já na primeira disciplina de programação que cursamos ao ingressar na universidade.

É importante que você leia este material com cuidado, executando cada uma das células de código e analisando seus resultados. Também experimente alterar os valores das variáveis aqui e ali, para compreender a relação delas com o resultado final. Procedendo desta maneira ao longo de todo o curso, você será capaz de criar e manipular blocos de nota Jupyter com facilidade.

1 Objetos e variáveis

Em Python, toda e qualquer informação que é manipulada em um programa está associada a um *rótulo* e um *objeto*. Um **objeto** por ser visto como uma região da memória que contém dados (o conteúdo) e informações adicionais sobre estes dados. Estas informações podem ser, por exemplo, o tipo do conteúdo armazenado e o endereço na memória aonde o objeto está localizado. Os objetos podem ser desde números inteiros até funções.

Os **rótulos** são identificadores compostos por um ou mais caracteres concatenados, sempre iniciando com uma letra ou sublinhado (_). Os demais caracteres podem ser números, letras ou mais sublinhados. É importante saber também que o Python faz a diferenciação entre letras minúsculas e minúsculas. Alguns exemplos válidos de rótulos são:

`rotulo0`, `Rotulo0`, `rotulo_composto`, `__ROTULO__`.

Nos exemplos acima, os rótulos `rotulo0` e `Rotulo0` serão considerados diferentes e, portanto, associados a objetos diferentes.

Recomenda-se sempre escolher rótulos que possam dar uma pista para quem lê o programa de qual a utilidade do objeto correspondente. Usar o rótulo `cp` para denotar "casos positivos" pode dificultar a leitura de seu programa por outra pessoa. Que tal simplesmente usar: `casos_positivos`? Seu programa ficará ainda mais inteligível se você utilizar uma regra para construção desses rótulos. As regras básicas para códigos em Python estão aqui.

Como a Python possui diversos identificadores pré-definidos (e.g., `class`, `continue`, `list`, `dict`, `True` e `False`), devemos tomar cuidado na criação de novos rótulos.

Na prática, é comum nos referirmos a um dado representado por um rótulo e seu objeto usando simplesmente o termo **variável**, como acontecerá de agora em diante. Isto facilitará muito o nosso diálogo.

1.1 Tipos de dados

Toda variável está associada a um determinado *tipo de dado*. Costumamos classificar os tipos de dados em: *primitivos* e *compostos*. Alguns dos tipos mais usados em Python são:

- Primitivos: `int`, `float`, `long`, `complex`
- Compostos: `str`, `list`, `tuple`, `set`, `dict`

As variáveis são criadas no momento em que atribuímos valores a elas usando o operador `=`. É exatamente neste momento que tanto o *tipo* quanto o *tamanho* de uma variável são definidos, de modo dinâmico. Por exemplo, ao efetuarmos as atribuições:

[1]:
```
n = 0
temp = 28.7
msg = "Determinante nulo."
```

criamos * uma variável de nome `n`, valor igual a 0 e tipo `int`. * uma variável de nome `temp`, valor igual a 28.7 (será?) e tipo `float`. * uma variável de nome `msg`, valor igual a `Determinante nulo.` e tipo `str`.

Podemos consultar os valores dessas variáveis escrevendo:

[2]:
```
n,temp,msg
```

[2]:
```
(0, 28.7, 'Determinante nulo.')
```

Já o tipo de uma variável pode ser consultado com a função `type` (- Ué!? Mas nem falamos de função ainda?!)

[3]:
```
type(n),type(temp),type(msg)
```

[3]:
```
(int, float, str)
```

Agora, o tamanho das variáveis dependerá do valor atribuído, dentre outros aspectos. O tamanho de uma variável é retornado em *bytes* pela função `getsizeof` do pacote `sys`.

[4]:
```
from sys import getsizeof
getsizeof(n),getsizeof(temp),getsizeof(msg)
```

[4]:
```
(24, 24, 67)
```

Pularei os detalhes sobre a caixa de comando acima, para focarmos no mais importante neste momento, o conceito de variável.

A saída acima indica que a `n` ocupa 24 bytes na memória. Olha o que acontece quando atribuímos 1 ao `n`:

[5]:
```
n = 1
getsizeof(n)
```

[5]:
```
28
```

Observe que foram adicionados 4 bytes ao tamanho anterior, resultando em 28 bytes. Na verdade, teremos $24+4k$ bytes toda vez que você tiver um inteiro positivo menor do que 2^{30k}, $k \geq 0$. Vejamos:

[6]:
```
getsizeof(0),getsizeof(1),getsizeof(2**30),getsizeof(2**60),getsizeof(2**90)
```

[6]:
```
(24, 28, 32, 36, 40)
```

O tipo `int` em Python 3 é ilimitado! Cada tipo de variável tem sua regra própria de dimensionamento.

1.2 Números

Os tipos intrínsecos (ou internos) de números mais utilizados em Python são: `int`, `float`, `bool` e `complex`. Por exemplo,

[7]:
```
a = 33
b = 1.78
c = True
d = 3 + 2j

type(a),type(b),type(c),type(d)
```

[7]:
```
(int, float, bool, complex)
```

Para estes números, estão definidas operações aritméticas e lógicas. Por exemplo,

[8]:
```
65 - a
```

[8]:
```
32
```

[9]:
```
b > 1.7 and a > 16
```

[9]:
```
True
```

[10]:
```
not c
```

[10]:
```
False
```

[11]:
```
d + a
```

[11]:
```
(36+2j)
```

O tipo do objeto resultante da operação será o "menor" dos tipos envolvidos na expressão, aquele suficiente para representar o resultado. Se lembrarmos da teoria de conjuntos, onde:

$$\{0,1\} \subset \mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$$

conseguiremos prever o tipo resultante de uma expressão.

1.3 Cadeias de caracteres

Uma cadeia de caracteres pode ser vista como um container imutável de caracteres alfanuméricos, delimitada por aspas simples ou duplas.

[12]:
```
narrador = "Então ele disse:"
narrador
```

[12]:
```
'Então ele disse:'
```

[13]:
```
fala = '- Parabéns, Cícero!'
fala
```

[13]:
```
'- Parabéns, Cícero!'
```

Podemos usar o operador + para concatenar cadeias de caracteres:

[14]:
```
texto = narrador + ' ' + fala
texto
```

[14]:
```
'Então ele disse: - Parabéns, Cícero!'
```

1.4 Listas

As listas em Python são objetos bastante versáteis. Elas são usadas para armazenar coleções de outros objetos (inclusive outras listas!). Uma lista é delimitada por colchetes, com seus elementos separados por vírgula. Seus elementos podem ser de tipos iguais (homogênea) ou diferentes (heterogênea). A Python não faz distinção.

[15]:
```
['K', 3.14, 1j]
B = ['K', 3.14, 1j]
A = [1, 1.41, B, 33]
A
```

[15]:
```
[1, 1.41, ['K', 3.14, 1j], 33]
```

Dizemos que o primeiro elemento de uma lista fica localizado na posição `0`, o segundo na posição `1`, e assim sucessivamente. A quantidade de elementos em uma lista pode ser obtida pela função `len`:

[16]:
```
len(A)
```

[16]:
```
4
```

Operações básicas em listas Vejamos algumas das operações mais úteis quando trabalhamos com listas. Podemos inserir um elemento no final de uma lista usando o método `append`:

[17]:
```
A = ['pedra','papel']
A.append('tesoura')
A
```

[17]:
```
['pedra', 'papel', 'tesoura']
```

A remoção de um elemento da lista é realizada com a função `remove`.

[18]:
```
A.remove('papel')
A
```

[18]:
```
['pedra', 'tesoura']
```

Esta função remove a primeira ocorrência da palavra `papel` em `A`.

Outra operação útil, é a concatenação de listas:

[19]:
```
A = [1, 2]
B = ['feijão', 'com', 'arroz']
A + B
```

[19]:
```
[1, 2, 'feijão', 'com', 'arroz']
```

Indexação e fatiamento As listas permitem ainda o uso de indexação, o que possibilita utilizá-las para representar arranjos lineares (vetores). Os índices iniciam em zero e vão até o número de elementos menos um.

[20]:
```
A = ['p','y','t','h','o','n']
A[0],A[5]
```

[20]:
```
('p', 'n')
```

Uma funcionalidade bastante útil das listas é a possibilidade de acessarmos blocos de elementos, técnica denominada de **fatiamento**. Extraímos do terceiro ao quarto elemento de `A` como a seguir:

[21]:
```
B = A[2:4]
B
```

[21]:
```
['t', 'h']
```

1.5 O segredo das atribuições em Python

Neste momento, é importante ressaltar o real significado de uma simples atribuição em Python. Quando escrevemos:

`a = b`

isto significa:

- copie o objeto associado ao identificador `b` para o objeto associado ao identificador `a`, quando `b` é um tipo **imutável** (e.g., `int`, `float`, `complex`, `str`, `tuple`, `range`)
- associe ao objeto correspondente ao identificador `b` ao identificador `a`, quando `b` é um tipo **mutável** (e.g., `list`, `set`, `dict`)

Analisemos o trecho abaixo:

[22]:
```
B = [5,7,11]
A = B
A[2] = 'foo'
A,B
```

[22]:
```
([5, 7, 'foo'], [5, 7, 'foo'])
```

Como tanto `a` quanto `b` estão associados ao mesmo objeto, pois listas são mutáveis, a alteração no valor de `a[2]` tem efeito em `b`. Portanto, aqui temos uma única lista, associada a dois identificadores distintos.

Agora, consideremos algo um pouco diferente.

[23]:
```
B = [5,7,11]
A = B[:]
A[2] = 'foo'
A,B
```

[23]:
```
([5, 7, 'foo'], [5, 7, 11])
```

Porque isto acontece? O segredo está na operação de fatiamento `a = b[:]`. Esta operação cria uma cópia do objeto associado a `b`. Portanto, o objeto ao qual `a` corresponde é diferente daquele de `b`. Aqui, de fato, temos duas listas distintas.

2 Sentenças e comentários

Uma *sentença* é um conjunto de uma ou mais instruções que o interpretador de Python pode executar. Por exemplo, cada linha a seguir é uma sentença:

[24]:
```
a = 2.5
b = 3.14159
c = a*b**2
c
```

[24]:
```
24.67396932025
```

Um *comentário* pode ser definido como uma sentença que não possui instrução associada, é uma instrução "vazia".

2.1 Quebra e continuação de sentenças

Podemos colocar várias sentenças em uma única linha, desde que estejam separadas por ponto-e-vírgula.

[25]:
```
a = 4; b = 2.5; c = -1; d = 7.2
a, b, c, d
```

[25]:
```
(4, 2.5, -1, 7.2)
```

Embora possível, a concatenação de sentenças em uma única linha geralmente prejudica a inteligibilidade.

No caso de sentenças longas, muitas vezes é melhor quebrá-las em diversas linhas utilizando o caractere de continuação \:

[26]:
```
d = a*(a - b)*(a - c) - \
    b*(b - a)*(b - c) + \
    c**a
d
```

[26]: 44.125

Agora, se a sentença possuir um par de parenteses, podemos quebrá-la em qualquer ponto no interior dos parenteses sem a necessidade de usarmos o caracter \.

[27]:
```
e = (1/3)*(a + 2*b +
           2*c + 2*d)
e
```

[27]: 31.75

2.2 Comentários

Tudo o que produzimos na vida, se não for possível de ser compreendido, deixa de ter utilidade. Quando programamos, normalmente, desejamos resolver um problema que possa levar a melhorias em um domínio específico. Por isso, é essencial nos esforçarmos para tornar nosso programa compreensível. Também não é para a gente sair explicando o significado de cada linha de código. Basta fazermos comentários explicativos em blocos chave. A inclusão desses comentários em códigos em Python pode ser feita de dois modos:

1. Digitando # (hashtag ou jogo da velha), todo o resto da linha à direita deste caractere será visto como um comentário e não será executado pelo interpretador.

[28]:
```
# Este é um exemplo de comentário.
# Abaixo, segue o código:
pi = 3.14159
pi
```

[28]: 3.14159

2. Digitando um par de aspas triplas, todo o texto contido no interior do par será um comentário. Este modo é mais indicado para comentários longos. Por exemplo:

[29]:
```
"""
Este é um exemplo de comentário longo.
Observe que aqui há duas sentenças, cada uma em uma linha distinta.
"""
r = 2.0
area = pi*(r**2)
area
```

[29]: 12.56636

3 Condicionais

Por padrão, o interpretador Python executa um conjunto de sentenças na ordem em que elas estão escritas, iniciando no topo do texto. Usamos o condicional `if` para alterar o fluxo natural do interpretador. A sintaxe completa do `if` é:

`if` < expressão booleana 1 >:
> < bloco 1 >

`elif` < expressão booleana 2 >:
> < bloco 2 >

`elif` < expressão booleana 3 >:
> < bloco 3 >

⋮

`elif` < expressão booleana n - 1 >:
> < bloco n - 1 >

`else`:
> < bloco n >

Ilustramos o uso do `if` com um teste de sinal para números reais.

[30]:
```
a = -1.35
if a > 0:
    resposta = "Positivo"
elif a < 0:
    resposta = "Negativo"
else:
    resposta = "Nulo"

print(resposta)
```

```
Negativo
```

Observação. Lembre-se que em Python a indentação é essencial!

4 Repetições

A utilidade dos computadores está principalmente no fato deles possuirem a capacidade de realizar tarefas repetitivas rapidamente e, o que é melhor, sem reclamar! :)

Python possui duas instruções que permitem a repetição de blocos de sentenças: `for` e `while`. Essas repetições são conhecidas como *laços*. Neste momento, nos restringiremos a explicar o uso do `for`.

A estrutura básica de um laço usando `for` é a seguinte.

`for` < iterador > in < container de objetos >:
> < bloco de sentenças >

onde `< iterator >` será um objeto usado para acessar cada elemento do `< container de objetos >`. Por exemplo,

[31]:
```
A = [1.2, 3.5, -2.9, 0.32, 1.87]
s = 0
for b in A:
    s = s + b
s
```

[31]: 3.99

No código acima, o que representa a variável `s`?

Como as listas permitem indexação iniciando do zero, podemos escrever o algoritmo anterior de outro modo.

[32]:
```
A = [1.2, 3.5, -2.9, 0.32, 1.87]
s = 0
for i in range(len(A)): # len(a) é igual ao número de elementos de 'a'
    s = s + A[i]
s
```

[32]: 3.99

Na versão acima, utilizamos as funções `range` e `len` para gerar uma lista de números inteiros. A função `len` retorna a quantidade de elementos em uma lista. A forma geral da função `range` é:

`range`(inicio, fim, passo)

Ela retorna uma classe do tipo `range` contendo números inteiros regularmente espaçados, iniciando em `inicio`, todos menores do que `fim`. O `passo` define a distância entre elementos consecutivos da lista. Seguem alguns exemplos:

[33]:
```
A = list(range(4))
B = list(range(3,7))
C = list(range(10,1,-2))
A,B,C
```

[33]: ([0, 1, 2, 3], [3, 4, 5, 6], [10, 8, 6, 4, 2])

4.1 Saídas prematuras

Nem sempre precisamos executar um laço do início ao fim. Quando estamos buscando um objeto dentro de uma lista, por exemplo, percorremos a lista até encontrarmos o objeto desejado. No momento em que ele é encontrado, podemos interromper o laço. Esta interrupção é feita com o `break`.

[34]:
```
a = 4
B = list(range(10,1,-2))
for x in B:
    if x == a:
        break
x
```

[34]: 4

Outra palavra-chave que é muito útil é o `continue`, utilizado para "pularmos" algumas iterações de um laço.

[35]:
```
a = 4
B = list(range(10,1,-2))
s = 0
for x in B:
    if x == a:
        continue
    s = s + x
s
```

[35]: 26

5 Funções

Uma função é um objeto que encapsula uma sequência de sentenças que podem ser executadas inúmeras vezes dentro de um programa. Ela pode ser definida em qualquer região do código fonte, mas sempre antes do ponto onde será utilizada.

A forma geral de uma função é:

`def` < nome da funcao >(< lista de argumentos >):
> < corpo da funcao >

Por exemplo, a função retorna o maior de dois números.

[36]:
```
def maior(a,b):
    if a > b:
        print(a)
    else:
        print(b)
```

[37]:
```
maior(3,2)
```

3

[38]:
```
maior(-1,-3)
```

-1

As funções podem ainda possuir uma valor de retorno. Para ilustrar isto, considere uma função $f\colon \mathbb{R} \to \mathbb{R}^2$, definida por $f(x) = (x, x^2)$. Em Python, podemos implementá-la assim:

[39]:
```
def f(x):
    return (x, x**2)
```

Vejamos o resultado de $f(3)$:

[40]:
```
f(3)
```

[40]:
```
(3, 9)
```

Observe que é necessário empregar a palavra-chave `return` para que f de fato retorne o valor calculado.

5.1 Passagem de parâmetros

Quando passamos argumentos para uma função, o que acontece é uma simples **atribuição**. Se o que passamos é um rótulo para um objeto *imutável* (e.g., `int`, `float`, `bool`, `str`, `tuple`), a função não tem como alterar o valor armazenado no objeto original, fora da função. Caso contrário, é possível atualizarmos de dentro da função o conteúdo de uma variável criada fora dela.

Definimos a função `asterisco` a seguir para ilustrar como fazer isso. Ela atribui o caractere `'*'` à posição k da lista `A`, sempre que k for menor do que `len(A)`.

[41]:
```
def asterisco(A,k):
    if k < len(A):
        A[k] = '*'
```

O resultado da `asterisco` quando aplicada a uma lista `B = [1, 2, 3]`, com $k = 0$, é o seguinte:

[42]:
```
B = ['1', '2','3']
asterisco(B,0)
B
```

[42]:
```
['*', '2', '3']
```

6 Exercício.

Agora, você deve aplicar o que aprendeu de Jupyter e Python para implementar um método que gere uma sequência de números inteiros bastante famosa na Matemática, a **sequência de Fibonacci**. Alguns números desta sequência são:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, \ldots$$

Podemos definir a sequência de Fibonacci $(F_n)_{n=0}^{\infty}$ usando recursividade! Primeiramente definimos $F_0 = 0$ e $F_1 = 1$, os quais chamamos de **casos base**. Agora, estabelecemos a seguinte relação recursiva:

$$F_n = F_{n-2} + F_{n-1}, \text{ para } n = 2, 3, 4, \ldots$$

Sua tarefa é completar a função `fib` abaixo de modo que ela retorne o valor de F_n, utilizando um algoritmo de sua escolha.

[43]:
```
def fib(n):
    """Retorna o valor n-ésimo número de Fibonacci.

    Argumento:
      n (int): número inteiro >= 0.

    Retorno:
      (int): valor do n-ésimo número de Fibonacci.
    """
    # Você deve começar a escrever o código de sua função a partir␣
 ↪daqui.
```

Sabendo que $F_{32} = 2178309$, execute a célula abaixo e analise o resultado.

[44]:
```
%%time
fib(32) == 2178309
```

```
CPU times: user 4 µs, sys: 1 µs, total: 5 µs
Wall time: 8.34 µs
```

[44]:
```
False
```

7 Saiba mais

- Como prometido, apresentamos aqui apenas uma pitada de Python. Vimos conceitos básicos e, ainda assim, de modo rasteiro. Para realmente aprender a programar em Python é necessário muito mais do que isso. É preciso praticar todos os dias!
- Quer entender mais sobre bits e bytes? Assista ao vídeo do Alexandre Meslin.
- Quer saber mais sobre as regras de dimensionamento de objetos em Python? Leia: https://stackoverflow.com/questions/449560/how-do-i-determine-the-size-of-an-object-in-python
- Diversos resultados em torno da sequência de Fibonacci estão catalogados na *The On-Line Encyclopedia of Integer Sequences*, no endereço: https://oeis.org/A000045.
- Achou a sua implementação de Fibonacci rápida? Dê uma olhada neste artigo aqui. Nele são apresentados doze algoritmos para calcular F_n. Será que o seu é o melhor?
- Para descontrair, assista ao vídeo de Arthur Benjamin, **A magia dos números de Fibonacci**, TED Talks

[45]:
```
from IPython.display import YouTubeVideo
YouTubeVideo(id='SjSHVDfXHQ4',width=600,height=300)
```

[45]:

UFCA | Centro de Ciências e Tecnologia