A Linguagem Python (uma introdução *muito* rápida)

Thiago Teixeira Santos
<thiago.santos@embrapa.br>

17 de novembro de 2017

1 A Linguagem Python

Esta seção apresenta uma **introdução sucinta** à linguagem Python. Pesquisadores com experiência prévia em programação procedural como Fortran, C ou MATLAB não deverão ter maiores dificuldades com o conteúdo a seguir, uma vez que ele define em Python conceitos familiares como atribuição de variáveis, controle de fluxo e iterações. Pesquisadores que não possuem conhecimentos prévios sobre programação de computadores podem buscar orientação no projeto Software Carpentry, destinado a ensinar diversas técnicas de computação a pesquisadores com formações acadêmicas variadas.

1.1 Tipos numéricos

Há quatro tipos numéricos em Python: inteiros simples (*plain integers* - ou apenas inteiro), inteiros longo (*long integers*), números em ponto flutuante (*floating points*) e complexos.

1.1.1 Inteiros

A diferença entre o inteiro e o inteiro longo é que o último possui *precisão infinita*, permitindo a representação de qualquer número em $\mathbb{Z}$ (na prática, há limites impostos pela memória do computador). Na maior parte do tempo, o programador utiliza inteiros simples, capazes de representar números inteiros no intervalo $[-n-1, n]$. O valor de n varia de acordo com o sistema (32 ou 64 bits), podendo ser consultado na variável `sys.maxint`:

```
In [2]: import sys

        sys.maxint

Out[2]: 9223372036854775807
```

Inteiros são representados e manipulados trivialvente:

```
In [3]: 1 + 1

Out[3]: 2
```

Como usual na maioria das linguagens procedurais, uma **atribuição de variável** é realizada utilizando-se o operador `=`. Podemos conferir o **tipo** da variável utilizando a função `type`:

```
In [4]: a = 4
        type(a)

Out[4]: int
```

O resto da divisão inteira pode ser obtido utilizando o operador %:

```
In [5]: 11 % 4

Out[5]: 3
```

Inteiros longos (precisão infinita) são representados acionando-se a letra `l` (minúscula ou maiúscual), ao final do inteiro:

```
In [6]: a = 32L
        a

Out[6]: 32L

In [7]: b = a * sys.maxint
        b

Out[7]: 295147905179352825824L

In [8]: c = b * sys.maxint
        c

Out[8]: 27222589353675077071167010490954400399 68L

In [9]: type(c)

Out[9]: long
```

1.1.2 Reais

Diferenciam-se os números reais dos inteiros com o uso do caractere ".", opcionalmente anexando-se casas decimais se houver:

```
In [10]: a = 2
         type(a)

Out[10]: int

In [11]: a = 2.
         type(a)

Out[11]: float

In [12]: a = 2.1
         type(a)

Out[12]: float
```

Informações sobre a representação dos números em ponto flutuante no sistema, como valores máximos e mínimos, podem sem obtidos com a variável `sys.float_info`:

```
In [13]: sys.float_info

Out[13]: sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308, min=2.2250738
```

1.1.3 Complexos

Números complexos tem sua parte imaginária definida pelo caractere "`j`". Os atributos `real` e `imag` permitem acesso direto às partes real e imaginária do número.

```
In [14]: a = 1.2 + 0.7j
         a

Out[14]: (1.2+0.7j)

In [15]: type(a)

Out[15]: complex

In [16]: a.real

Out[16]: 1.2

In [17]: a.imag

Out[17]: 0.7
```

1.2 Operações matemáticas

Além das operações aritméticas de adição, subtração, multiplicação e divisão, Python fornece uma maneira simples de definir **potenciação**, utilizando a notação "`**`":

```
In [18]: 2**3

Out[18]: 8
```

Quanto à divisão, vale lembra a diferença entre dividir números inteiros e reais:

```
In [19]: 3/2

Out[19]: 1

In [20]: 3./2

Out[20]: 1.5
```

1.3 *Containers*

Todas as variáveis em Python, até mesmo as variáveis numéricas básicas, são objetos. Uma categoria especial de objetos em Python são os *containers*, capazes de armazenar outros objetos, como listas, tuplas e dicionários.

1.3.1 Sequências

Listas Listas são a sequência mais comum e mais utilizada em Python. Listas podem ser criadas diretamente utilizando-se colchetes "`[]`", como no exemplo abaixo contendo oito objetos *string*:

```
In [21]: L = ['vermelho', 'azul', 'verde', 'amarelo', 'ciano', 'magenta', 'branco', 'preto']
         type(L)

Out[21]: list

In [22]: L

Out[22]: ['vermelho', 'azul', 'verde', 'amarelo', 'ciano', 'magenta', 'branco', 'preto']
```

Elementos da lista podem ser acessados por seu **índice**, lembrando que, de modo similar a Fortran e C mas diferentemente de MATLAB, o primeiro elemento é indexado por 0:

```
In [23]: L[0]

Out[23]: 'vermelho'

In [24]: L[2]

Out[24]: 'verde'
```

Um recurso útil na indexação em Python é que índices negativos podem ser utilizados para acessar a lista em **ordem reversa**. Por exemplo, -1 pode ser utilizado para acessar o último elemento da lista:

```
In [25]: L[-1]

Out[25]: 'preto'

In [26]: L[-2]

Out[26]: 'branco'
```

Outro recurso útil é chamado de *slicing*. Utilizando a notação `início:fim:passo`, podemos obter listas que são subsequências da lista inicial:

```
In [27]: L[2:6]

Out[27]: ['verde', 'amarelo', 'ciano', 'magenta']

In [28]: L[2:]

Out[28]: ['verde', 'amarelo', 'ciano', 'magenta', 'branco', 'preto']

In [30]: L[1:6:2]

Out[30]: ['azul', 'amarelo', 'magenta']
```

Listas podem conter tipos diferentes de objetos:

```
In [31]: L = [3, -200.7, 3+5j, 'hello']
         L

Out[31]: [3, -200.7, (3+5j), 'hello']
```

O método `pop` pode ser utilizado para obter e simultaneamente retirar um objeto da lista. Seu comportamente padrão (sem argumentos) é remover o último elemento da lista:

```
In [32]: x = L.pop()
         x

Out[32]: 'hello'

In [33]: L

Out[33]: [3, -200.7, (3+5j)]
```

A posição do elemento-alvo pode ser informada:

```
In [34]: x = L.pop(0)
         x

Out[34]: 3

In [35]: L

Out[35]: [-200.7, (3+5j)]
```

Os métodos `append` e `extend` podem ser utilizados para adicionar elementos à lista. O primero adiciona um único objeto como elemento da lista. Já `extend` adiciona a lista cada um dos elementos de uma segunda lista passada como argumento:

```
In [36]: L

Out[36]: [-200.7, (3+5j)]

In [37]: L.append(42)
         L

Out[37]: [-200.7, (3+5j), 42]

In [38]: L.extend([3, 5, 'ueba!'])
         L

Out[38]: [-200.7, (3+5j), 42, 3, 5, 'ueba!']
```

Se uma lista for utilizada como argumento da função `append`, ela será inserida como um único elemento:

```
In [39]: L.append([1,2,3])
         L

Out[39]: [-200.7, (3+5j), 42, 3, 5, 'ueba!', [1, 2, 3]]
```

Tuplas Tuplas são **listas imutáveis**. Elementos podem ser lidos a partir das tuplas e até mesmo *slicing* pode ser utilizado para gerar novas tuplas. Porém, não é possível alterar uma tupla de forma alguma após sua criação.

```
In [39]: ponto = (23, 542)

In [40]: ponto

Out[40]: (23, 542)

In [41]: ponto[0]

Out[41]: 23

In [42]: x, y = ponto
         print x
         print y

23
542
```

1.3.2 Dicionários

Dicionários são tabelas para armazenamento eficiente de pares *chave, valor*. Eles são implementados através de eficientes **tabelas de espalhamento** (*hash tables*).

```
In [43]: tel = {'emmanuelle': 5752, 'sebastian': 5578}

In [44]: tel

Out[44]: {'emmanuelle': 5752, 'sebastian': 5578}

In [45]: tel.keys()

Out[45]: ['sebastian', 'emmanuelle']

In [46]: tel.values()

Out[46]: [5578, 5752]

In [47]: tel['sebastian']

Out[47]: 5578
```

Um nova entrada pode ser adicionada ao dicionário simplesmente atribuindo-se um valor a uma nova chave:

```
In [48]: tel['thiago'] = 5823
         tel
```

```
Out[48]: {'emmanuelle': 5752, 'sebastian': 5578, 'thiago': 5823}
```

- Iteração em dicionários

```
In [49]: for k in tel.keys():
             print 'Fale com %s no ramal %d' % (k, tel[k])
```

```
Fale com sebastian no ramal 5578
Fale com thiago no ramal 5823
Fale com emmanuelle no ramal 5752
```

1.4 Controle de fluxo

1.4.1 if/elif/else

```
In [50]: if 2+2 == 4:
             print 'OK, o Universo está bem'
```

```
OK, o Universo está bem
```

```
In [51]: a = 10

         if a == 1:
             print 'É pouco'
         elif a == 2:
             print 'É bom'
         else:
             print 'É demais!'
```

```
É demais!
```

1.4.2 for/range

```
In [52]: for i in range(5):
             print i
```

```
0
1
2
3
4
```

```
In [53]: for word in ('legal', 'poderoso', 'legível'):
             print 'Python é %s' % word
```

```
Python é legal
Python é poderoso
Python é legível
```

```
In [54]: for i, word in enumerate(('legal', 'poderoso', 'legível')):
             print '%dº: Python é %s' % (i, word)
```

```
0º: Python é legal
1º: Python é poderoso
2º: Python é legível
```

1.4.3 while/break/continue

```
In [55]: z = 1 + 1j

         while abs(z) < 100:
             if z.imag == 0:
                 break
             z = z**2 + 1
         z
```

```
Out[55]: (-134+352j)
```

1.5 Compreensão de listas

Compreensão de listas é um dos recursos mais interessantes da linguagem Python. Esse recurso possibilita a criação dinâmica de novas listas a partir de objetos iteráveis, sendo mais eficiente que o uso de um laço `for` e produzindo código mais legível. Considere o exemplo abaixo:

```
In [40]: sqr = [i**2 for i in range(10)]
         sqr
```

```
Out[40]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

A semântica do código acima é simples: `sqr` é uma lista contendo i^2 para cada i na sequência `range(10)`, que no caso é o intervalo $[0,9]$. Considere outro exemplo:

```
In [41]: doc = ['Hello', 'Embrapa', 'Informática']
         wlen = [len(word) for word in doc]
         wlen
```

```
Out[41]: [5, 7, 12]
```

Aqui, `wlen` é a lista contendo o comprimento de cada palavra na sequência `doc`. Usada corretamente, a compreensão de listas pode produzir códigos elegantes de fácil compreensão por parte de outros pesquisadores. Note como o primeiro exemplo corresponde diretamente a expressão matemática:

$i^2, \forall i \in [0,9]$.