Aula 1B - Fundamentos de Python

Gustavo Oliveira[1] e Andrea Rocha[1]

[1]Departamento de Computação Científica / UFPB

Junho de 2020

1 Fundamentos de *Python* para Computação Científica

1.1 Python como calculadora de bolso

Nesta seção mostraremos como podemos fazer cálculos usando Python como uma calculadora científica. Como Python é uma *linguagem interpretada*, ela funciona diretamente como um ciclo *LAI (Ler-Avaliar-Imprimir)*. Isto é, ela lê uma instrução que você escreve, avalia essa instrução interpretando-a e imprimindo (ou não) um valor.

1.1.1 Aritmética

Os símbolos são os seguintes:

operação	símbolo
adição	+
subtração	-
multiplicação	*
divisão	/
potenciação	**

Exemplo: calcule o valor de

$$4/3 - 2 \times 3^4$$

[2]:
```
4/3 - 2*3**4
```

[2]:
```
-160.66666666666666
```

1.1.2 Expressões numéricas

Comutatividade, associatividade, distributividade e agrupamento de números por meio de parênteses, colchetes ou chaves são escritos em Python usando apenas parênteses.

Exemplo: calcule o valor de

$$2 - 98\{7/3\,[2 \times (3 + 5 \times 6) - (2^3 + 2^2) - (2^2 + 2^3)] - (-1)\}$$

[3]:
```
2 - 98*(7/3*(2*(3+5*6)-(2**3+2**2)-(2**2+2**3))-(-1))
```

[3]:
```
-9700.0
```

Note que a legibilidade da instrução acima não está boa. Podemos incluir espaços para melhorá-la. Espaços **não alteram** o resultado do cálculo.

[4]:
```
2 - 98*( 7/3*( 2*( 3 + 5*6 ) - ( 2**3 + 2**2 ) - ( 2**2 + 2**3 ) ) - (-1) )
```

[4]:
```
-9700.0
```

Um dos princípios do Zen do Python é *a legibilidade conta*, ou seja, vale a pena aperfeiçoar sua maneira de escrever um código computacional de maneira que outros possam entendê-lo sem dificuldades quando o lerem.

Parênteses não pareados (abrir, mas não fechar) produzirão um erro.

[5]:
```
2 - 98*( 7/3
```

```
  File "<ipython-input-5-ef2d20508391>", line 1
    2 - 98*( 7/3
                ^
SyntaxError: unexpected EOF while parsing
```

O erro diz que houve uma terminação não esperada da instrução. Em geral, quando digitamos um parêntese de abertura, o Jupyter adiciona o de fechamento automaticamente.

[6]:
```
2 - 98*(7/3)
```

[6]:
```
-226.66666666666669
```

1.1.3 Números inteiros e fracionários

Em Python, números inteiros e fracionários são escritos de maneira distinta através do ponto decimal.

Exemplo: Calcule o valor de

$$2 + 3$$

[7]:
```
2 + 3
```

[7]:
```
5
```

Exemplo: Calcule o valor de

$$2,0 + 3$$

[8]:
```
2.0 + 3
```

[8]:
```
5.0
```

O valor é o mesmo, mas em Python, os dois números acima possuem naturezas diferentes. Essa "natureza" é chamada *tipo*. Falaremos disso mais tarde. Por enquanto, veja o seguinte:

[9]:
```
type(5)
```

[9]:
```
int
```

[10]:
```
type(5.0)
```

[10]:
```
float
```

As duas palavras acima indicam que o **tipo** do valor 5 é `int` e o do valor 5.0 é `float`. Isto significa que no primeiro caso temos um número inteiro, mas no segundo caso temos um número em *ponto flutuante*. Números em ponto flutuante imitam o conjunto dos números reais. Veremos mais exemplos disso ao longo do curso.

Exemplo: Calcule o valor de

$$5.0 - 5$$

[11]:
```
5.0 - 5
```

[11]:
```
0.0
```

Observe a resposta no caso anterior. A diferença entre um número em ponto flutuante e um número inteiro resulta em um número em ponto flutuante! Isto é verdadeiro para as versões 3.x (onde x é um número maior ou igual a 0) da linguagem Python, mas não o é para versões anteriores da linguagem.

Vale mencionar este detalhe para seu conhecimento. Entretanto, usaremos Python 3 neste curso e você não deverá ter problemas com isso no caminho.

Usar ponto ou usar vírgula? Em nosso sistema numérico, a vírgula (`,`) é quem separa a parte inteira da parte fracionária de um número decimal. Em Python, este papel é desempenhado pelo ponto (`.`). Daqui para a frente, sempre que não houver ambiguidade de notação, usaremos o ponto e não a vírgula para representar números fracionários em exemplos, exercícios ou explicações.

Exemplo: calcule o valor de

$$5.1 - 5$$

[12]:
```
5.1 - 5
```

[12]:
```
0.09999999999999964
```

Observe o cálculo anterior... O resultado deveria ser 0.1, não? O que aconteceu?!

Isto se deve a um conceito chamado *precisão de máquina*. Não entraremos em detalhe nisto neste curso, mas é suficiente que você saiba que números no computador não são exatos como na Matemática tradicional.

Um computador, por mais rápido e inteligente que possa parecer, é incapaz de representar a infinidade dos números reais. Às vezes, ele *aproximará* resultados. Portanto, não se preocupe com isso. A conta parece errada e, de fato, está! Porém, este erro é tão pequeno que praticamente não afetará suas contas significativamente.

1.1.4 Divisão inteira

Podemos realizar a operação de divisão inteira quando quisermos um quociente inteiro. Para isso, utilizamos o símbolo // (divisão inteira).

Exemplo: calcule o valor de

$$5/2$$

[13]:
```
5/2
```

[13]:
```
2.5
```

Exemplo: calcule o valor da **divisão inteira**

$$5/2$$

[14]:
```
5//2
```

[14]:
```
2
```

O algoritmo da divisão e restos Você deve se lembrar do algoritmo da divisão. Um número D (dividendo), quando dividido por outro número d (divisor), resulta em um quociente q e um resto r nulo se a divisão for exata, ou um resto r diferente de zero, se a divisão não for exata. Ou seja, o algoritmo diz o seguinte:

$$D = d \times q + r$$

Em Python, podemos descobrir o valor r de uma divisão inexata diretamente por meio do símbolo %, chamado de **operador módulo**.

Exemplo: determine o resto da divisão

$$7/2$$

[15]:
```
7 % 2
```

[15]:
```
1
```

De fato, $5 = 2 \times 3 + 1$

A partir disso, podemos então verificar números inteiros pares e ímpares com o operador %.

[16]:
```
5 % 2
```

[16]:
```
1
```

[17]:
```
6 % 2
```

[17]:
```
0
```

Com efeito, 5 é ímpar, pois, ao ser dividido por 2, retorna resto igual a 1, e 6 é par, pois, sendo exatamente divisível por 2, retorna resto 0.

1.1.5 Porcentagem

Então, se % serve para calcular o resto de uma divisão, como calcular porcentagem?

Bem, não há um símbolo especial para o cálculo da porcentagem. Ele deve ser feito dividindo o número em questão por 100 da maneira usual.

Exemplo: Quanto é 45% de R$ 43,28?

[18]:
```
45/100*43.28
```

[18]:
```
19.476000000000003
```

Veja que poderíamos também realizar esta conta da seguinte forma:

[19]:
```
0.45*43.28
```

[19]:
```
19.476000000000003
```

Ou da forma:

[20]:
```
45*43.28/100
```

[20]:
```
19.476000000000003
```

Esta última não seria tão literal quanto à conta original, porém, os três exemplos mostram que a multiplicação e a divisão são equivalentes em **precedência**. Tanto faz neste caso realizar primeiro a multiplicação e depois a divisão, ou vice-versa. A única ressalva é o segundo exemplo, no qual, na verdade, não foi o computador quem dividiu 45 por 100.

Precedência de operações Assim como na Matemática, Python possui precedências em operações.

Quando não há parênteses envolvidos, multiplicação e divisão são realizadas antes de adições e subtrações.

Exemplo: calcule $4 \times 5 + 3$

[21]:
```
4*5 + 3
```

[21]:
```
23
```

Exemplo: calcule $4/5 + 3$

[22]:
```
4/5 + 3
```

[22]:
```
3.8
```

Quando houver parênteses, eles têm precedência.

Exemplo: calcule $4 \times (5 + 3)$

[23]:
```
4*(5+3)
```

[23]:
```
32
```

Em regra, operações são executadas da esquerda para a direita e do par de parênteses mais interno para o mais externo.

Exemplo: calcule $2 - 10 - 3$

[24]:
```
# primeiramente, 2 - 10 = - 8 é calculado;
# depois - 8 - 3 = -11
2 - 10 - 3
```

[24]:
```
-11
```

Exemplo: calcule $2 - (10 - 3)$

[25]:
```
# primeiramente, (10 - 3) = 7 é calculado;
# depois  2 - 7 = -5
2 - (10 - 3)
```

[25]:
```
-5
```

1.1.6 Comentários

Note que acima descrevemos passos executados pelo interpretador Python para calcular as expressões numéricas. Porém, fizemos isso em uma célula de código e não houve nenhuma interferência no resultado. Por quê? Porque inserimos *comentários*.

Comentários em linha São utilizados para ignorar tudo o que vier após o símbolo # naquela linha.

[26]:
```
# isto aqui é ignorado
```

[27]:
```
2 + 3 # esta linha será calculada
```

[27]:
```
5
```

A instrução abaixo resulta em erro, pois `2 +` não é uma operação completa.

[28]:
```
2 + # 3
```

```
  File "<ipython-input-28-50842ae2b13f>", line 1
    2 + # 3
          ^
SyntaxError: invalid syntax
```

A instrução abaixo resulta em 2, pois `+ 3` está comentado.

[29]:
```
2 # + 3
```

[29]:
```
2
```

Exemplo: qual é o valor de

$$(3-4)\hat{}2+(7/2-2-(3+1))?$$

[30]:
```
""" ORDEM DE OPERAÇÕES
primeiramente, (3 + 1) = 4 é calculado :: parêntese mais interno
em seguida, 7/2 = 3.5 :: divisão mais interna
em seguida, 3.5 - 2  = 1.5 :: primeira subtração mais interna
em seguida, 1.5 - 4  = -2.5 :: segunda subtração e resolve parêntese externo
em seguida, (3 - 4) = -1 :: parêntese
em seguida, (-1)**2 = 1 :: potenciação, ou duas multiplicações
em seguida, 1 + (-2.5) = -1.5 :: última soma
"""

# Expressão
(3 - 4)**2 + (7/2 - 2 - (3 + 1))
```

[30]:
```
-1.5
```

Docstrings Em Python, não há comentários em bloco, mas podemos usar *docstrings* quando queremos comentar uma ou múltiplas linhas. Basta inserir um par de três aspas simples (`'''`) ou duplas (`"`). Aspas simples são também chamadas de "plicas" (`'...'`).

Por exemplo,

```
'''
Isto aqui é
um comentário em bloco e
será ignorado pelo
interpretador se vier
seguido de outra instruções.
'''
```

e

```
"""
Isto aqui é
um comentário em bloco e
será ignorado pelo
interpretador se vier
seguido de outra instruções.
"""
```

possuem o mesmo efeito prático, porém existem recomendações de estilo para usar docstrings. Não discutiremos esses tópicos aqui.

Se inseridas isoladamente numa célula de código aqui, produzem uma saída textual.

Cuidado! plicas não são apóstrofes!

[31]:
```
"""
Isto aqui é
um comentário em bloco e
será ignorado pelo
interpretador se vier
seguido de outra instruções.
"""
```

[31]:
```
'\nIsto aqui é \num comentário em bloco e\nserá ignorado pelo \ninterpretador se
vier \nseguido de outra instruções.\n'
```

1.1.7 Omitindo impressão

Podemos usar um ; para omitir a impressão do último comando executado na célula.

[32]:
```
2 + 3; # a saída não é impressa na tela
```

[33]:
```
2 - 3
1 - 2; # nada é impresso pois o último comando vem seguido por ;
```

[34]:
```
"""
Docstring múlipla
não impressa
""";
```

[35]:
```
'''Docstring simples omitida''';
```

1.2 Variáveis, atribuição e reatribuição

Em Matemática, é muito comum usarmos letras para substituir valores. Em Python, uma variável é um nome associado a um local na memória do computador. A idéia é similar ao endereço de seu domicílio.

Exemplo: se $x = 2$ e $y = 3$, calcule $x + y$

[37]:
```
x = 2
y = 3
x + y
```

[37]: 5

Acima, x e y são variáveis. O símbolo = indica que fizemos uma atribuição.

Uma reatribuição ocorre quando fazemos uma nova atribuição na mesma variável. Isto é chamade de *overwriting*.

[38]:
```
x = 2 # x tem valor 2
y = 3 # y tem valor 3
x = y # x tem valor 3
x
```

[38]: 3

Exemplo: A área de um retângulo de base b e altura h é dada por $A = bh$. Calcule valores de A para diferentes valores de b e h.

[39]:
```
b = 2 # base
h = 4 # altura
A = b*h
A
```

[39]: 8

[40]:
```
b = 10 # reatribuição em b
A = b*h # reatribuição em A, mas não em h
A
```

[40]: 40

[41]:
```
h = 5 # reatribuição em h
b*h # cálculo sem atribuição em A
```

[41]: 50

[42]:
```
A # A não foi alterado
```

[42]: 40

Variáveis são *case sensitive*, isto é, sensíveis a maiúsculas/minúsculas.

[43]:
```
a = 2
A = 3
```

```
a - A
```

[43]: -1

```
A - a
```

[44]: 1

1.2.1 Atribuição por desempacotamento

Podemos realizar atribuições em uma única linha.

```
b,h = 2,4
```

```
b
```

[46]: 2

```
h
```

[47]: 4

1.2.2 Imprimindo com `print`

`print` é uma função. Aprenderemos um pouco sobre funções mais à frente. Por enquanto, basta entender que ela funciona da seguinte forma:

```
print(A) # imprime o valor de A
```

3

```
print(b*h) # imprime o valor do produto b*h
```

8

Com `print`, podemos ter mais de uma saída na célula.

```
b, h = 2.1, 3.4
print(b)
print(h)
```

2.1
3.4

Podemos inserir mais de uma variável em print separando-as por vírgula.

```
print(b, h)
```

2.1 3.4

[52]:
```
x = 0
y = 1
z = 2
print(x, y, z)
```

```
0 1 2
```

[53]:
```
print(x, x + 1, x + 2)
```

```
0 1 2
```

1.3 Caracteres, letras, palavras: strings

Em Python, escrevemos caracteres entre plicas ou aspas.

[54]:
```
'Olá!'
```

[54]:
```
'Olá!'
```

[55]:
```
"Olá!"
```

[55]:
```
'Olá!'
```

[56]:
```
'Olá, meu nome é Python.'
```

[56]:
```
'Olá, meu nome é Python.'
```

Podemos atribuir caracteres a variáveis.

[57]:
```
a = 'a'
A = 'A'
print(a)
print(A)
```

```
a
A
```

[58]:
```
area1 = 'Matemática'
area2 = 'Estatística'
print(area1, 'e', area2)
```

```
Matemática e Estatística
```

Nomes de variáveis não podem iniciar por números ou conter caracteres especiais ($, #, ?, !, etc)

[59]:
```
1a = 'a' # inválido
```

```
  File "<ipython-input-59-85b25870aa03>", line 1
1a = 'a' # inválido
```

```
      ^
SyntaxError: invalid syntax
```

[60]:
```
a1 = 'a1' # válido
a2 = 'a2' # válido
```

Podemos usar `print` concatenando variáveis e valores.

[61]:
```
b = 20
h = 10
print('Aprendo', area1, 'e', area2)
print('A área do retângulo é', b*h)
```

```
Aprendo Matemática e Estatística
A área do retângulo é 200
```

Em Python, tudo é um "objeto". Nos comandos abaixo, temos objetos de três "tipos" diferentes.

[62]:
```
a = 1
x = 2.0
b = 'b'
```

Poderíamos também fazer:

[63]:
```
a, x, b = 1, 2.0, 'b' # modo menos legível!
```

Ao investigar essas variáveis (objetos) com `type`, veja o que temos:

[64]:
```
type(a) # verifica o "tipo" do objeto
```

[64]:
```
int
```

[65]:
```
type(x)
```

[65]:
```
float
```

[66]:
```
type(b)
```

[66]:
```
str
```

`str` (abreviação de *string*) é um objeto definido por uma cadeia de 0 ou mais caracteres.

[67]:
```
nenhum = ''
nenhum
```

[67]:
```
''
```

[68]:
```
espaco = ' '
espaco
```

[68]:
```
' '
```

Exemplo: Calcule o valor da área de um círculo de raio $R = 3$ e imprima o seu valor usando print e strings. Assuma o valor de $\pi = 3.145$.

[69]:
```
R = 3
pi = 3.145
A = pi*R**2
print('A área do círculo é: ', A)
```

```
A área do círculo é:  28.305
```

1.4 Tipos de dados: `int`, `float` e `str`

Até o momento, aprendemos a trabalhar com números inteiros, fracionários e sequencias de caracteres.

Em Python, cada objeto possui uma "família". Chamamos essas famílias de "tipos".

Fazendo um paralelo com a teoria dos conjuntos em Matemática, você sabe que $\mathbb{Z}$ representa o *conjunto dos números inteiros* e que $\mathbb{R}$ representa o *conjunto dos números inteiros*.

Basicamente, se `type(x)` é `int`, isto equivale a dizer que $x \in \mathbb{Z}$. Semelhantemente, se `type(y)` é `float`, isto é "quase o mesmo" que dizer $y \in \mathbb{R}$. Porém, neste caso, não é uma verdade absoluta para todo número y.

Futuramente, você aprenderá mais sobre "ponto flutuante". Então, é mais correto dizer que $y \in \mathbb{F}$, onde $\mathbb{F}$ é seria o conjunto de todos os números em *ponto flutuante*. No final das contas, um número de $\mathbb{F}$ faz uma "aproximação" para um número de $\mathbb{R}$.

No caso de `str`, é mais difícil estabelecer uma notação similar, mas podemos criar exemplos.

Exemplo: Considere o conjunto $A = \{s \in \mathbb{S} \ : s \text{ possui apenas duas vogais}\}$, onde $\mathbb{S}$ é o conjunto de palavras formadas por 2 letras.

Todo elemento s desse conjunto pode assumir um dos seguintes valores:

`aa`, `ae`, `ai`, `ao`, `au`,

`ea`, `ee`, `ei`, `eo`, `eu`,

`ia`, `ie`, `ii`, `io`, `iu`,

`oa`, `oe`, `oi`, `oo`, `ou`,

`ua`, `ue`, `ui`, `uo`, `uu`.

Apesar de apenas algumas terem significado na nossa língua, como é o caso de `ai`, `ei`, `oi` e `ui`, que são interjeições, `eu`, que é um pronome e `ou`, que é uma conjunção, existem 25 *anagramas*.

Então, poderíamos escrever:

[70]:
```
s1, s2, s3, s4, s5  = 'aa', 'ae', 'ai', 'ao', 'au'
s6, s7, s8, s9, s10  = 'ea', 'ee', 'ei', 'eo', 'eu'
s11, s12, s13, s14, s15  = 'ia', 'ie', 'ii', 'io', 'iu'
s16, s17, s18, s19, s20  = 'oa', 'oe', 'oi', 'oo', 'ou'
s21, s22, s23, s24, s25  = 'oa', 'oe', 'oi', 'oo', 'ou'
```

Os 25 anagramas acima foram armazenados em 25 variáveis diferentes.

[71]:
```
print(s3)
print(s11)
print(s24)
```

```
ai
ia
oo
```

1.4.1 *Casting*

Uma das coisas legais que podemos fazer em Python é alterar o tipo de um dado para outro. Essa operação é chamada de *type casting*, ou simplesmente *casting*.

Para fazer *casting* de `int`, `float` e `str`, usamos funções de mesmo nome.

[72]:
```
float(25) # 25 é um inteiro, mas float(25) é fracionário
```

[72]:
```
25.0
```

[73]:
```
int(34.21) # 34.21 é fracionário, mas int(34.21) é um "arredondamento" para um␣
↪inteiro
```

[73]:
```
34
```

[74]:
```
int(5.65) # o arredondamento é sempre para o inteiro mais próximo "para baixo"
```

[74]:
```
5
```

[75]:
```
int(-6.6)
```

[75]:
```
-6
```

O *casting* de um objeto 'str' composto de letras com 'int' é inválido.

[76]:
```
int('a')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
```

```
    <ipython-input-76-3c9f902a4b53> in <module>
----> 1 int('a')


    ValueError: invalid literal for int() with base 10: 'a'
```

O *casting* de um 'int' ou 'float' com 'str' formada como número é válido.

[77]:
```
str(2)
```

[77]:
```
'2'
```

[78]:
```
str(3.14)
```

[78]:
```
'3.14'
```

O *casting* de um 'str' puro com 'float' é inválido.

[79]:
```
float('a')
```

```
    ---------------------------------------------------------------------------
    ValueError                                Traceback (most recent call last)

    <ipython-input-79-688063d46f27> in <module>
----> 1 float('a')


    ValueError: could not convert string to float: 'a'
```

1.4.2 Concatenação

Outra coisa legal que podemos fazer em Python é a *concatenação* de objetos `str`. A concatenação pode ser feita de modo direto como uma "soma" (usando +) ou acompanhada por *casting*.

[80]:
```
s21 + s23
```

[80]:
```
'oaoi'
```

[81]:
```
s1 + s3 + s5
```

[81]:
```
'aaaiau'
```

[82]:
```
'Casting do número ' + str(2) + '.'
```

[82]:
```
'Casting do número 2.'
```

[83]:
```
str(1) + ' é inteiro' + ',' + ' ' + 'mas ' + str(3.1415) + ' é fracionário!'
```

[83]:
```
'1 é inteiro, mas 3.1415 é fracionário!'
```

Podemos criar concatenações de `str` por repetição usando multiplicação (`*`) e usar parênteses para formar as mais diversas "montagens".

[84]:
```
(str(s3) + ',')*2 + s3 + '...'
```

[84]:
```
'ai,ai,ai...'
```

[85]:
```
x = 1.0
'Usando 0.1, incrementamos ' + str(x) + ' para obter ' + str(x + 0.1) + '.'
```

[85]:
```
'Usando 0.1, incrementamos 1.0 para obter 1.1.'
```

[93]:
```
print(5*s20 + 10*s2 + 5*s17 + 5*('.') + 'YEAH! :)')
```

```
ouououououaeaeaeaeaeaeaeaeaeaeoeoeoeoeoeoe...YEAH! :)
```

1.5 Módulos e importação

Um mecânico, bombeiro hidráulico ou eletricista sempre anda com uma caixa de ferramentas contendo ferramentas essenciais, tais como alicate, chave de fenda e parafusadeira. Cada ferramenta possui uma função bem definida. Porém, quando esses profissionais deparam com uma tarefa que nenhuma ferramenta de sua caixa é capaz de executar, é necessário buscar por outra ferramenta para resolver o problema.

Podemos entender Python de modo similar. A linguagem fornece uma estrutura mínima de ferramentas que pode ser expandida. Por exemplo, até agora aprendemos a somar e a subtrair, mas ainda não sabemos calcular a raiz quadrada de um número.

Fazemos isto com a importação de funções que "habitam" em *módulos*. Sempre que precisarmos de algo especial em nossa maleta de ferramentas de *data science*, devemos pesquisar por alguma solução existente ou criar nossa própria solução. Neste curso, não vamos nos aprofundar no tema de criar nossas próprias soluções, o que se chama *customização*. Você aprenderá mais sobre isto no devido tempo. Usaremos coisas já construídas para ganharmos tempo.

Módulos são como gavetas em um escritório ou equipamentos em uma oficina. Vários módulos podem ser organizados juntos para formar um *pacote*. Então, imagine que você precisa substituir o cabo RJ45 do seu desktop que seu *pet* roeu e o deixou sem internet no final de semana. Além dos terminais e de muita paciência, você precisará de um alicate de crimpagem para construir um cabo novo.

O alicate de crimpagem é um alicate especial, assim como a chave phillips é um tipo de chave especial. Se você tiver todas essas ferramentas na sua casa e for organizado, a primeira coisa que você fará é ir até a gaveta onde aquele tipo de ferramenta está guardada. Em seguida, você pegará a ferramenta especializada.

Em Python, quando precisamos de algo que desempenha um papel específico, realizamos a importação de um módulo inteiro, de um submódulo deste módulo ou de apenas um objeto do módulo. Há muitas maneiras de fazer isso.

Para importar um módulo inteiro, podemos usar a sintaxe:

```
import nomeDoModulo
```

Para importar um submódulo, podemos usar a sintaxe:

```
import nomeDoModulo.nomeDoSubmodulo
```

ou a sintaxe

```
from nomeDoModulo import nomeDoSubmodulo
```

Caso queiramos usar uma função específica, podemos usar

```
from nomeDoModulo import nomeDaFuncao
```

ou

```
from nomeDoModulo.nomeDoSubmodulo import nomeDaFuncao
```

se a função pertencer a um submódulo.

Existe uma forma muito eficiente de acessar os objetos de um módulo específico usando um *alias* (pseudônimo). O *alias* é um nome substituto para o módulo. Outra maneira de enxergar esse tipo de importação é pensar em uma chave que abre um cadeado. Este tipo de importação usa a sintaxe:

```
import nomeDoModulo as nomeQueEuQuero
```

Basicamente, esta sentença diz: *"importe o módulo `nomeDoModulo` como `nomeQueEuQuero`"*. Isto fará com que `nomeQueEuQuero` seja um pseudônimo para o módulo que você quer acessar. Entretanto, isto faz mais sentido quando o pseudônimo é uma palavra com menos caracteres. Por exemplo,

```
import nomeDoModulo as nqeq
```

Neste caso, `nqeq` é um pseudônimo mais curto. Ao longo do curso usaremos pseudônimos que já se tornaram praticamente um padrão na comunidade Python.

Como último exemplo de importação, considere a sintaxe:

```
from nomeDoModulo import nomeDoSubmodulo as nds
```

Neste exemplo, estamos criando um pseudônimo para um submódulo.

1.5.1 O módulo `math`

O módulo `math` é uma biblioteca de funções matemáticas. O que é uma função?

Em Matemática, uma função é como uma máquina que recebe uma matéria-prima e entrega um produto. Se a matéria-prima é x, o produto é y e a função é f, então, $y = f(x)$. Logo, para cada valor de entrada x, um valor de saída y é esperado.

Neste capítulo, já lidamos com a função `print`. O que ela faz?

Ela recebe um conteúdo, o valor de entrada (chamado de *argumento*), e o produto (valor de saída) é a impressão do conteúdo.

A partir de agora, vamos usar o módulo `math` para realizar operações matemáticas mais específicas.

Importaremos o módulo `math` como:

[98]:
```
import math as mt
```

Note que, aparentemente, nada aconteceu. Porém, este comando permite que várias funções sejam usadas em nosso *espaço de trabalho* usando a "chave" `mt` para abrir o "cadeado" `math`.

Para ver uma lista das funções existentes, escreva `mt.` e pressione a tecla `<TAB>`.

O número neperiano (ou número de Euler) e é obtido como:

[100]:
```
mt.e
```

[100]:
```
2.718281828459045
```

O valor de π pode ser obtido como:

[102]:
```
mt.pi
```

[102]:
```
3.141592653589793
```

Note, no entanto, que e e π são números irracionais. No computador, eles possuem um número finito de casas decimais! Acima, ambos possuem 16 dígitos.

Exemplo: calcule a área de um círculo de raio $r = 3$.

[108]:
```
r = 3 # raio
area = mt.pi*r**2 # area
print('A área é',area) # imprime com concatenação
```

```
A área é 28.274333882308138
```

A raiz quadrada de um número x, $\sqrt{x}$, é calculada pela função `sqrt` (abreviação de "square root")

[111]:
```
x = 3 # note que x é um 'int'
mt.sqrt(x) # o resultado é um 'float'
```

[111]:
```
1.7320508075688772
```

[113]:
```
mt.sqrt(16) # 16 é 'int', mas o resultado é 'float'
```

[113]:
```
4.0
```

Exemplo: calcule o valor de $\sqrt{\sqrt{\pi} + e + \left(\frac{3}{2}\right)^y}$, para $y = 2.1$.

[115]:
```
mt.sqrt( mt.sqrt( mt.pi ) + mt.e + 3/2**2.1 ) # espaços dão legibilidade
```

[115]:
```
2.2782691726433835
```

O logaritmo de um número b na base a é dado por $\log_a b$, com $a > 0$, $b > 0$ e $a \neq 1$.

- Quando $a = e$ (base neperiana), temos o *logaritmo natural* de b, denotado por $\ln b$.
- Quando $a = 10$ (base 10), temos o *logaritmo em base 10* de b, denotado por $\log_{10} b$, ou simplesmente $\log b$.

Em Python, algum cuidado deve ser tomado com a função logaritmo.

- Para calcular $\ln b$, use `log(b)`.
- Para calcular $\log b$, use `log10(b)`.
- Para calcular $\log_2 b$, use `log2(b)`.
- Para calcular $\log_a b$, use `log(b,a)`.

Pelas duas últimas colocações, vemos que `log(b,2)` é o mesmo que `log2(b)`.

Vejamos alguns exemplos:

[118]:
```
mt.log(2) # isto é ln(2)
```

[118]:
```
0.6931471805599453
```

[119]:
```
mt.log10(2) # isto é log(2) na base 10
```

[119]:
```
0.3010299956639812
```

[120]:
```
mt.log(2,10) # isto é o mesmo que a anterior
```

[120]:
```
0.30102999566398114
```

[123]:
```
mt.log2(2) # isto é log(2) na base 2
```

[123]:
```
1.0
```

Exemplo: se $f(x) = \dfrac{\ln(x+4) + \log_3 x}{\log_{10} x}$, calcule o valor de $f(e) + f(\pi)$.

[129]:
```
x = mt.e # x = e

fe = ( mt.log(x + 4) + mt.log(x,3) ) / ( mt.log10(x) ) # f(e)

x = mt.pi # reatribuição do valor de x

fpi = ( mt.log(x + 4) + mt.log(x,3) ) / ( mt.log10(x) ) # f(pi)

print('O valor é', fe + fpi)
```

```
O valor é 12.53225811727087
```

No exemplo anterior, espaços foram acrescentados para tornar os comandos mais legíveis.

1.6 Introspecção

Podemos entender mais sobre módulos, funções e suas capacidades examinando seus componentes e pedindo ajuda sobre eles.

Para listar todos os objetos de um módulo, use `dir(nomeDoModulo)`.

Para pedir ajuda sobre um objeto, use a função `help` ou um ponto de interrogação após o nome `?`.

[130]:
```
dir(mt) # lista todas as funções do módulo math
```

[130]:
```
['__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'acos',
 'acosh',
 'asin',
 'asinh',
 'atan',
 'atan2',
 'atanh',
 'ceil',
 'copysign',
 'cos',
 'cosh',
 'degrees',
 'e',
 'erf',
 'erfc',
 'exp',
 'expm1',
 'fabs',
 'factorial',
 'floor',
 'fmod',
 'frexp',
 'fsum',
 'gamma',
 'gcd',
 'hypot',
 'inf',
 'isclose',
 'isfinite',
 'isinf',
 'isnan',
 'ldexp',
```

```
 'lgamma',
 'log',
 'log10',
 'log1p',
 'log2',
 'modf',
 'nan',
 'pi',
 'pow',
 'radians',
 'remainder',
 'sin',
 'sinh',
 'sqrt',
 'tan',
 'tanh',
 'tau',
 'trunc']
```

[131]:
```
help(mt.pow) # ajuda sobre a função 'pow' do módulo 'math'
```

```
Help on built-in function pow in module math:

pow(x, y, /)
    Return x**y (x to the power of y).
```

Como vemos, a função `pow` do módulo `math` serve para realizar uma potenciação do tipo x^y.

[132]:
```
mt.pow(2,3) # 2 elevado a 3
```

[132]:
```
8.0
```

Exemplo: considere o triângulo retângulo com catetos de comprimento $a = \frac{3}{4}\pi\, m$ e $b = \frac{2}{e}\, m$. Qual é o comprimento da hipotenusa c?

[136]:
```
'''
Resolução pelo Teorema de Pitágoras
c = sqrt( a**2 + b**2 )
'''
a = 3./4. * mt.pi # 3. e 4. é o mesmo que 3.0 e 4.0
b = 2./mt.e
c = mt.sqrt( a**2 + b**2 )
print('O valor da hipotenusa é:', c, 'm')
```

```
O valor da hipotenusa é: 2.4683989970341536 m
```

O mesmo cálculo acima poderia ser resolvido com apenas uma linha usando a função `hypot` do módulo `math`.

[139]:
```
c = mt.hypot(a,b) # hypot calcula a hipotenusa
print('O valor da hipotenusa é:', c, 'm')
```

```
O valor da hipotenusa é: 2.4683989970341536 m
```

Exemplo: Converta o ângulo de 270° para radianos.

1 radiano (*rad*) é igual a um arco de medida r de uma circunferência cujo raio mede r. Isto é, $r = 1\,rad$. Uma vez que 180° corresponde a meia-circunferência, πr, então, $\pi\,rad = 180°$. Por regra de três, podemos concluir que $x°$ equivale a $\frac{x}{180°}\pi\,rad$.

[142]:
```
ang_graus = 270 # valor em graus
ang_rad = ang_graus/180*mt.pi # valor em radianos
print(ang_rad)
```

```
4.71238898038469
```

Note que em

```
ang_graus/180*mt.pi
```

a divisão é executada antes da multiplicação porque vem primeiro à esquerda. O parênteses não é necessário neste caso.

Poderíamos chegar ao mesmo resultado diretamente com a função `radians` do módulo `math`.

[144]:
```
mt.radians?
```

```
[0;31mSignature:[0m [0mmt[0m[0;34m.[0m[0mradians[0m[0;34m([0m[0mx[0m[0;34m,[0m [0;34m/[0m[0;34m)
[0;31mDocstring:[0m Convert angle x from degrees to radians.
[0;31mType:[0m      builtin_function_or_method
```

[145]:
```
mt.radians(ang_graus)
```

[145]:
```
4.71238898038469
```

1.6.1 Arredondamento de números fracionários para inteiros

Com `math`, podemos arredondar números fracionários para inteiros usando a regra "para cima" (teto) ou "para baixo" (chão).

- Use `ceil` (abreviatura de *ceiling*, ou "teto") para arredondar para cima;
- Use `floor` ("chão") para arredondar para baixo;

[154]:
```
mt.ceil(3.42)
```

[154]:
```
4
```

[155]:
```
mt.floor(3.42)
```

[155]: 3

[156]:
```
mt.ceil(3.5)
```

[156]: 3

[158]:
```
mt.floor(3.5)
```

[158]: 3

1.7 Números complexos

Números complexos são muito importantes no estudo de fenômenos físicos envolvendo sons, frequencias e vibrações. Na Matemática, o conjunto dos números complexos é definido como

$\mathbb{C} = \{z = a + bi \ : \ a, b \in \mathbb{R} \text{ e } i = \sqrt{-1}\}$. O valor i é o *número imaginário*.

Em Python, os números complexos são objetos do tipo `complex` e são escritos na forma

```
z = a + bj
```

ou

```
z = a + bJ
```

O símbolo `j` (ou `J`) quando vem acompanhado de um `int` ou `float` define a parte imaginária do número complexo.

[160]:
```
3 - 2j
```

[160]: (3-2j)

[162]:
```
type(3 - 2j) # o número é um complex
```

[162]: complex

[165]:
```
4J # J (maiúsculo)
```

[165]: 4j

[166]:
```
type(4j)
```

[166]: complex

Se `j` ou `J` são colocados isoladamente, significarão uma variável. Caso a variável não esteja definida, um erro de indefinição resultará.

[167]:
```
j
```

```
NameError                                 Traceback (most recent call last)

<ipython-input-167-3eedd8854d1e> in <module>
----> 1 j


NameError: name 'j' is not defined
```

1.7.1 Parte real, parte imaginária e conjugados

Números complexos também podem ser diretamente definidos como `complex(a,b)` onde `a` é a parte real e `b` é a parte imaginária.

[169]:
```
complex(6.3,9.8)
```

[169]:
```
(6.3+9.8j)
```

As partes real e imaginária de um `complex` são extraídas usando as funções `real` e `imag`, nesta ordem.

[172]:
```
z = 6.3 + 9.8j
z.real
```

[172]:
```
6.3
```

[173]:
```
z.imag
```

[173]:
```
9.8
```

O conjugado de `z` pode ser encontrado com a função `conjugate()`.

[175]:
```
z.conjugate()
```

[175]:
```
(6.3-9.8j)
```

1.7.2 Módulo de um número complexo

Se $Re(z)$ e $Im(z)$ forem, respectivamente, a parte real e a parte imaginária de um número complexo, o *módulo* de z é definido como

$$|z| = \sqrt{[Re(z)]^2 + [Im(z)]^2}$$

Exemplo: se $z = 2 + 2i$, calcule o valor de $|z|$.

Podemos computar esta quantidade de maneiras distintas. Uma delas é:

[177]:
```
(z.real ** 2 + z.imag ** 2) ** 0.5
```

[177]:
```
11.650321883965267
```

Entretanto, podemos usar a função `abs` do Python. Esta função é uma função predefinida pertencente ao *core* da linguagem.

[179]:
```
abs(z)
```

[179]:
```
11.650321883965267
```

A função `abs` também serve para retornar o "valor absoluto" (ou módulo) de números reais. Lembremos que o módulo de um número real x é definido como

$$|x| = \begin{cases} x, & \text{se } x \geq 0 \\ -x, & \text{se } x < 0 \end{cases}$$

[183]:
```
help(abs)
```

```
Help on built-in function abs in module builtins:

abs(x, /)
    Return the absolute value of the argument.
```

[181]:
```
abs(-3.1)
```

[181]:
```
3.1
```

[182]:
```
abs(-mt.e)
```

[182]:
```
2.718281828459045
```

1.8 Zen do Python: o estilo "pythônico" dos códigos

O Python tem um estilo de programação próprio popularmente chamado de *Zen do Python*, que advoga por princípios de projeto. O Zen do Python foi elaborado por Tim Peters e foi documentado no [PEP 20]. Em língua portuguesa, os princípios seriam traduzidos da seguinte forma:

- Bonito é melhor que feio.
- Explícito é melhor que implícito.
- Simples é melhor que complexo.
- Complexo é melhor que complicado.
- Linear é melhor do que aninhado.
- Esparso é melhor que denso.
- Legibilidade conta.
- Casos especiais não são especiais o bastante para quebrar as regras.

- Ainda que praticidade vença a pureza.
- Erros nunca devem passar silenciosamente.
- A menos que sejam explicitamente silenciados.
- Diante da ambiguidade, recuse a tentação de adivinhar.
- Deveria haver um — e preferencialmente só um — modo óbvio para fazer algo.
- Embora esse modo possa não ser óbvio a princípio a menos que você seja holandês.
- Agora é melhor que nunca.
- Embora nunca frequentemente seja melhor que já.
- Se a implementação é difícil de explicar, é uma má ideia.
- Se a implementação é fácil de explicar, pode ser uma boa ideia.
- Namespaces são uma grande ideia — vamos ter mais dessas!

Um dos mais importantes é **deveria haver um — e preferencialmente só um — modo óbvio para fazer algo**. Isto quer dizer que códigos devem ser escritos de modo "pythônico", isto é, seguindo o estilo natural da linguagem. Você pode sempre lembrar desses princípios com a seguinte instrução:

```
import this
```