MC102 — Funções, Objetos e Classes

Rafael C. S. Schouery
rafael@ic.unicamp.br

Universidade Estadual de Campinas

Atualizado em: 2023-04-18 13:53

Função na Matemática

O que é uma função na matemática?

- Ex: $f(x) = x$, $f(x) = 2x + 3$, $f(x) = \sqrt{x}$, ...
- É uma relação entre dois conjuntos X e Y
- Para cada $x \in X$, temos um único $y \in Y$ relacionado ($f(x) = y$)
- Escrevemos $f \colon X \to Y$
- Note que x não precisa ser um único valor
  - Ex: $f(x_1, x_2) = x_1 + x_2$, $f(a, b, c) = a \cdot b \cdot c$, ...
  - Isso é, X pode ser o produto cartesiano de outros conjuntos
- Note que y também não precisa ser um único valor
  - Ex: $f(x_1, x_2) = (2x_1, 3x_2)$, $f(x) = (x, x^2, x^3)$, ...
  - Isso é, Y pode ser o produto cartesiano de outros conjuntos
  - Mas, no final das contas o resultado é um único vetor...

Informalmente, f nos diz como calcular $y = f(x)$ a partir de x

- Como obter uma saída a partir de uma entrada

Função na Pogramação

Na programação, o conceito de função é bem parecido

- Temos um conjunto de dados de entrada
  - Mesmo papel do X na função $f : X \to Y$
  - São chamados de parâmetros da função
- Temos as instruções de como calcular uma resposta
- A resposta calculada (saída) é o nosso "$y = f(x)$"

Exemplos com Pseudocódigo

$f(x) = x^2$

```
1 Quadrado(x)
2     Devolva x * x
3
4 Leia n
5 Imprima Quadrado(n)
```

$f(x) = |x|$

```
1 Absoluto(x)
2     Se x >= 0
3         Devolva x
4     Senão
5         Devolva -x
```

$f(x) =$ soma dos dígitos na base 10 de x

```
1 SomaDosDígitos(x)
2     soma = 0
3     Enquanto x > 0
4         soma = soma + x % 10
5         x = x / 10 # Divisão inteira
6     Devolva soma
```

Exemplos com Python

$f(x) = x^2$

```
1 def quadrado(x):
2     return x * x     # ou x ** 2
3
4 n = int(input())     # int e input também são funções!
5 print(quadrado(n))   # print também!
```

$f(x) = |x|$

```
1 def absoluto(x):
2     if x >= 0:       # função pode ter if/elif/else
3         return x
4     else:
5         return -x
```

$f(x) =$ soma dos dígitos na base 10 de x

```
1 def soma_dos_digitos(x):
2     soma = 0
3     while x > 0:     # pode ter while e for também
4         soma = soma + x % 10
5         x = x // 10
6     return soma
```

Usando funções no Python

Definindo a função, i.e., informando o Python que:

- a função existe
- qual o seu nome
- quais são seus parâmetros

```
1 def nome_da_funcao(parametro_1, parametro_2, ..., parametro_n):
2     #instruções para computar o resultado
3     return resultado
```

Chamando a função, i.e., pedindo que seja executada

- A execução segue para as instruções da função
- E depois retorna para onde estava

```
1 calculado = nome_da_funcao(valor_1, valor_2, ..., valor_n)
```

Observações:

- O valor passado para o parâmetro pode vir de uma constante, variável, ou expressão
- A ordem dos valores é importante!

Exemplo de um código completo

```
1  def le_lista(n):
2      lista = []
3      for i in range(n):
4          lista.append(float(input()))
5      return lista
6
7  def soma_valores(lista):
8      soma = 0
9      for x in lista:
10         soma += x
11     return soma
12
13 n = int(input())
14 lista = le_lista(n)
15 print(soma_valores(lista))
```

Nas linhas:

- 1 a 5 definimos uma função chamada `le_lista`
- 7 a 11 definimos uma função chamada `soma_valores`
- 13 a 15 chamamos essas funções para somar os números

Vamos simular esse código!

Exercícios

1. Faça uma função que acha o maior entre dois números
2. Faça uma função que verifica se um número é primo
3. Faça uma função que recebe uma lista e devolve uma nova lista invertida
4. Faça uma função que devolve a lista dos divisores de um número

Por que usar funções?

Usamos funções para

- Evitar repetição de código
- Reutilizar o código de outras pessoas
- Permitir que outros reutilizem o nosso código
- Deixar o programa mais fácil de entender
- Deixar o programa mais fácil de *debuggar*
- Criar conjuntos de funções (bibliotecas) úteis
- Entre muitas outras coisas

O uso de funções é parte fundamental da programação!

Exemplo

```
1  def primo(p):
2      k = 2
3      while k * k <= p:
4          if p % k == 0:
5              return False
6          k = k + 1
7      return True
8
9  p = int(input("Entre com p: "))
10 q = int(input("Entre com q: "))
11
12 if primo(p) and primo(q):
13     print("Ambos são primos")
14 else:
15     print("Pelo menos um deles não é primo")
```

Vantagens:

- Podemos chamar a função várias vezes
- Outra pessoa poderia usar a função `primo`
- Outra pessoa poderia implementar a função `primo`
- O código é mais "fácil" de ler

Funções que "não" devolvem valor

Uma função não precisa ter o comando `return`

Ex:

```
1 def imprime(lista):
2     for x in lista:
3         print(x)
```

Essa função não precisa devolver nada...

Mas, não é bem assim...

```
1 def imprime(lista):
2     for x in lista:
3         print(x)
4
5 valor = imprime([1, 2, 3, 4])
6 print(valor)
```

Será impresso, em cada linha, `1`, `2`, `3`, `4` e `None`

O tipo `NoneType`

O tipo `NoneType` tem um único valor, o `None`

- O `None` representa o nada...
- A ideia é que não é um número, não é uma string, etc.

É algo bastante comum em linguagens de programação

- Em outras linguagens pode chamar: `NULL`, `nil`, entre outros

Toda função de Python devolve algum valor

- Nem que esse valor seja `None`
- Se não há `return`, `None` é devolvido

Um cuidado!

Se executarmos esse código:

```
1  n = int(input())
2  lista = le_lista(n)
3  print(soma_valores(lista))
4
5  def le_lista(n):
6      lista = []
7      for i in range(n):
8          lista.append(float(input()))
9      return lista
10
11 def soma_valores(lista):
12     soma = 0
13     for x in lista:
14         soma += x
15     return soma
```

Temos o seguinte erro após digitar o valor de `n`

```
1 Traceback (most recent call last):
2   File "cuidado.py", line 2, in <module>
3     lista = le_lista(n)
4 NameError: name 'le_lista' is not defined
```

A função ainda não havia sido definida!

Outras informações

Uma função pode ter zero parâmetros

- Não recebe nada de entrada, mas tem saída...
- Ex: função que lê um número

Uma função pode chamar outras funções

- Na verdade, pode chamar a si mesmo!
- Veremos mais sobre isso no futuro!

É importante escolher bons nomes para as funções

- Um nome que descreva bem o que ela faz
- Mas tente criar um nome razoavelmente curto

Exercícios

Um número n é *perfeito* se n é a soma dos seus divisores próprios

- Ex: $6 = 1 + 2 + 3$

Exercício: Faça uma função que, dado n, decide se n é perfeito ou não.

Variáveis

O que é uma variável?

- Um lugar da memória
- Para o qual demos um nome

Podemos ter novas variáveis dentro das funções

Ex:

```
1 def soma_dos_digitos(x):
2     soma = 0
3     while x > 0:     # pode ter while e for também
4         soma = soma + x % 10
5         x = x // 10
6     return soma
```

`soma` é o que chamamos de variável local

- Ela existe apenas dentro da função
- E perde seu valor quando a função termina

Variáveis locais

Um exemplo:

```
1 def funcao(x):
2     y = 2 * x
3     return y
4
5 z = funcao(10)
6 print(z)
7 print(y)
```

O que é impresso por esse código?

```
1 20
2 Traceback (most recent call last):
3   File "vars1.py", line 7, in <module>
4     print(y)
5 NameError: name 'y' is not defined
```

y não pode ser acessada na linha 7

Variáveis globais

Outro exemplo:

```
1 def imprime():
2     print(z)
3
4 z = 10
5 imprime()
6 print(z)
```

O que é impresso por esse código?

```
1 10
2 10
```

z pode ser acessada na linha 2

- z é uma variável global
- pode ser acessada em qualquer função

Começando a confusão...

Mais um exemplo:

```
1 def imprime():
2     z = 8
3     print(z)
4
5 z = 10
6 imprime()
7 print(z)
```

O que é impresso por esse código?

```
1 8
2 10
```

O que aconteceu?

- O Python criou uma variável local chamada z
- Que não é a mesma variável global chamada z
- Elas podem ter valores diferentes!
- Não importa que elas tenham o mesmo nome!

Resolvendo

"Corrigindo" o exemplo anterior:

```
1 def imprime():
2     global z
3     z = 8
4     print(z)
5
6 z = 10
7 imprime()
8 print(z)
```

O que é impresso por esse código?

```
1 8
2 8
```

Dizemos para o Python que queremos usar a variável global z

Variáveis Locais e Globais

Variáveis Locais:

- São definidas dentro da função
  - Na primeira atribuição
- Só existem dentro da função
  - Dizemos que o escopo da variável é a função
- Podem ter o mesmo nome que variáveis globais
- Note que um parâmetro é uma variável local

Variáveis Globais:

- São definidas fora de qualquer função
- Podem ser escritas dentro das funções
  - Mas é necessário usar o `global`
  - Melhor devolver o valor do que alterar diretamente
- Podem ser lidas dentro das funções
  - Não precisa usar o `global`
  - Mas não pode ter uma variável local com o mesmo nome
  - Não usar o `global` pode levar a bugs!
- Idealmente não são manipuladas diretamente pelas funções

Escopo

O escopo de um nome (de variável, de função, etc)

- É a região do programa onde esse nome é válido
- Isto é, onde esse nome pode ser acessado
- Vimos o escopo global
  - Variável é acessível em qualquer parte do programa
- e o escopo local
  - Variável é acessível apenas dentro da função onde foi criada
- temos também o escopo *built-in*
  - funções como `int`, `input`
- mas também temos o escopo *enclosing*

Escopo *enclosing*

O seguinte código é válido em Python (e imprimi 30):

```
1 def f(x):
2     a = 10
3     def g(y):
4         print(a * y)
5     g(x + 1)
6
7 f(2)
```

- `g` é uma função local de `f`
- `a` e `x` são variáveis locais de `f`
- `y` é variável local de `g`
- `a` não é variável local de `g` e não é variável global...
  - mas é acessível em `g`
  - está *enclosing*

Objetos, Classes e Métodos

O que as variáveis armazenam?

- Objetos de um certo tipo (ou classe)

As classes definem métodos que podem ser utilizados pelos objetos

- Funções que acessam ou modificam os objetos

Exemplo:

- `l = [1, 2, 3]` cria um objeto do tipo `list`
- `append` é um método de `list` que adiciona um elemento ao final da lista
- ou seja, podemos escrever `l.append(4)`

Em breve aprenderemos a criar nossas próprias classes!

Funções e Listas

Qual o resultado desse código?

```
1 def soma_um(x):
2     x = x + 1
3
4 x = 10
5 soma_um(x)
6 print(x)
```

E deste?

```
1 def soma_um(lista):
2     #len(lista) diz quanto elementos há na lista
3     for i in range(len(lista)):
4         lista[i] = lista[i] + 1
5
6 lista = [1, 2, 3]
7 soma_um(lista)
8 print(lista)
```

A função altera a lista!

- Também alteraria se fizéssemos `append`
- Mais sobre isso em breve

Documentando funções

Além dos comentários normais, devemos usar `docstrings`

- Uma string que começa e termina com `"""`
- Sempre na primeira linha da função
- E que diz o que a função faz

Ex:

```
1 def soma_dos_digitos(x):
2     """Calcula a soma dos digitos do número x na base 10."""
3     soma = 0
4     while x > 0:
5         soma = soma + x % 10
6         x = x // 10
7     return soma
```

Isso permite usar a função `help` (e é usado pelos editores)

```
1 >>> help(soma_dos_digitos)
2 Help on function soma_dos_digitos in module exemplo_docstring:
3
4 soma_dos_digitos(x)
5     Calcula a soma dos digitos do número x na base 10.
```

Docstrings de várias linhas

Em geral, precisamos de várias linhas na documentação

```
1  def soma_dos_digitos(x):
2      """Calcula a soma dos digitos do número x na base 10.
3
4      Funciona apenas para números inteiros não negativos.
5
6      Parâmetros:
7      x -- número inteiro positivo
8      """
9      soma = 0
10     while x > 0:
11         soma = soma + x % 10
12         x = x // 10
13     return soma
```

Convenção de Estilo do Python (PEP 257)

- Primeira linha diz o que a função faz brevemente
- Deve haver uma linha em branco após a primeira linha
- Terminamos com """ sozinho na última linha

Outros detalhes

É possível definir um valor padrão para um parâmetro

Ex:

```
1 def soma(x, y = 2, z = 7):
2     return x + y + z
3
4 print(soma(3))        # imprime 3 + 2 + 7 = 12
5 print(soma(3, 4))     # imprime 3 + 4 + 7 = 14
6 print(soma(3, 4, 2)) # imprime 3 + 4 + 2 = 9
```

Se você quiser passar um parâmetro na posição i

- Precisa passar todos os parâmetros anteriores
- Mesmo os com valores padrão

Outros detalhes

É possível usar o nome do parâmetro na chamada da função

Ex:

```
1 def soma(x, y=2, z=7):
2     return x + y + z
3
4
5 print(soma(x=10))             # imprime 3 + 2 + 7 = 12
6 print(soma(x=3, y=4))         # imprime 3 + 4 + 7 = 14
7 print(soma(x=3, z=2))         # imprime 3 + 2 + 2 = 7
8 print(soma(x=3, y=4, z=2))    # imprime 3 + 4 + 2 = 9
```

Outros detalhes

Você até pode misturar os dois

- Isto é, ter parâmetros posicionais e nominais
- Porém, precisa começar com os parâmetros posicionais

Ex:

```
1  def soma(x, y=2, z=7):
2      return x + y + z
3
4
5  print(soma(10))                 # imprime 3 + 2 + 7 = 12
6  print(soma(x=10))               # imprime 3 + 2 + 7 = 12
7  print(soma(3, y=4))             # imprime 3 + 4 + 7 = 14
8  print(soma(x=3, y=4))           # imprime 3 + 4 + 7 = 14
9  print(soma(3, z=2))             # imprime 3 + 2 + 2 = 7
10 print(soma(x=3, z=2))           # imprime 3 + 2 + 2 = 7
11 print(soma(x=3, y=4, z=2))      # imprime 3 + 4 + 2 = 9
12 print(soma(3, 4, z=2))          # imprime 3 + 4 + 2 = 9
```

Existem formas de forçar parâmetros serem apenas posicionais ou apenas nominais

Exercícios

1. Faça uma função que, dado um número `n` e um valor `v` (por padrão, valendo `0`), cria uma lista de `n` posições com cada posição valendo `v`. Documente sua função.
2. Faça uma função que, dada uma lista `l` e um valor `v` (por padrão, valendo `0`), modifica todas as entradas da lista para `v`. Documente sua função.

Pilha de Chamadas

Ex: um código que calcula $\sum_{i=1}^{k} \frac{1}{i}$ (com bug...)

```
1  def divisao(x, y):  # apenas para fins didáticos
2      return x / y
3
4  def soma(k):
5      s = 0
6      for i in range(k):
7          s += divisao(1, i)
8      return s
9
10 s = soma(10)
11 print(s)
```

O que acontece quando executamos o código?

```
1 Traceback (most recent call last):
2   File "pilha.py", line 12, in <module>
3     s = soma(10)
4   File "pilha.py", line 8, in soma
5     s += divisao(1, i)
6   File "pilha.py", line 2, in divisao
7     return x / y
8 ZeroDivisionError: division by zero
```

Pilha de Chamadas

```
1 Traceback (most recent call last):
2   File "pilha.py", line 12, in <module>
3     s = soma(10)
4   File "pilha.py", line 8, in soma
5     s += divisao(1, i)
6   File "pilha.py", line 2, in divisao
7     return x / y
8 ZeroDivisionError: division by zero
```

O Python sabe qual linha fez qual chamada de função

- Isso gera o que chamamos de pilha
  - Mais sobre pilhas em MC202 — Estrutura de Dados
- Pense em uma pilha (de pratos)
- Quando uma função é chamada, ela vai em cima da atual
- Sabemos em que linha o problema ocorreu
  - Mas note que o bug está em outra linha!
  - A linha 8 chamou incorretamente `divisao`

Vamos debuggar o código anterior!

Exceções

O erro que vimos é o que chamamos de uma `exception`

- O seu programa termina assim que acontece a exceção
- Com o Python imprimindo a pilha de execução
- Mas nem sempre é isso que queremos...

Exemplo:

```
1 n = int(input("Entre com n: "))
2
3 if n % 2 == 0:
4     print(n, "é par")
5 else:
6     print(n, "é impar")
```

E se o usuário digitar um número errado?

```
1 Entre com n: 3.4
2 Traceback (most recent call last):
3   File "exception1.py", line 1, in <module>
4     n = int(input("Entre com n: "))
5 ValueError: invalid literal for int() with base 10: '3.4'
```

Precisamos lidar com as exceções!

try...except

Vamos

- tentar executar o código (try)
  - Um bloco de código onde pode aparecer uma exceção
- e lidar com exceções (except)
  - O que fazer se a exceção ocorrer

Código corrigido

```
1  def le_numero(mensagem):
2      while True:
3          try:
4              n = int(input(mensagem))
5              return n
6          except ValueError:
7              print("O valor digitado não é válido")
8
9
10 n = le_numero("Entre com n: ")
11 if n % 2 == 0:
12     print(n, "é par")
13 else:
14     print(n, "é impar")
```

Observações sobre `try`

Você pode capturar várias exceções no mesmo `except`

- `except (RuntimeError, TypeError, NameError):`

Você pode ter vários `except`s:

```
1 except OSError as err: #err contém as informações do erro
2     ...
3 except ValueError:
4     ...
5 except:
6     ...
```

Inclusive o último é genérico, serve para qualquer exceção

- o que pode ocultar bugs no seu código, cuidado!

Após todos os `except`s, você pode ter um `else:`

- O que fazer se nenhuma exceção ocorrer

Após todos os `except`s, você pode ter um `finally:`

- Executado independentemente de ter exceção
- Sempre é a última coisa a ser feita

Exemplo

```
1  try:
2      print("antes da divisão")
3      a = x / y
4      print("depois da divisão")
5  except ZeroDivisionError:
6      print("divisão por zero")
7  except:
8      print("erro")
9  else:
10     print("sem erro")
11 finally:
12     print("terminei")
```

- O que acontece se `x == 1` e `y == 0`?
- O que acontece se `x == 1` e `y == 1`?
- O que acontece se a variável `x` não existir?

Exemplo de Mau Uso de except

Um erro bobo, mas difícil de achar

```
1  def soma(lista):
2      s = 0
3      for k in lista:
4          s += k
5      return s
6
7  try:
8      numeros = [1, 2, 3, 4]
9      print(soma(numero))
10 except:
11     print("Código executado com erro")
```

Provavelmente eu ficaria procurando erro na função `soma`

- Ao invés de perceber que escrevi `numero` e não `numeros`
- Imagine isso é um código muito maior...

Por isso evitamos usar o `except` genérico

Levantando erros

Você pode levantar exceções com `raise`

```
1  def le_numero_positivo():
2      n = int(input("Digite um número: "))
3      if n < 0:
4          raise ValueError("Número negativo")
5
6
7  def le_lista_de_positivos(n):
8      lista = []
9      for i in range(n):
10         lista[i] = le_numero_positivo()
11     return lista
12
13
14 try:
15     lista = le_lista_de_positivos(5)
16     print(lista)
17 except ValueError as e:
18     print(e)
```

O erro pode ser capturado em qualquer parte da pilha de execução

Bibliotecas

O conceito de bibliotecas de código é muito importante

- A ideia é compartilhar código já escrito
- De uma maneira organizada e documentada
- Para que outros programadores possam reutilizar

Existem bibliotecas dos mais variados tipos:

- Para lidar com imagens
- Para criar sites dinâmicos
- Para criar jogos
- Para ler informações em determinados formatos
- Para acessar conteúdos na internet

O programador consegue focar na sua tarefa

- e não reinventar a roda

Algumas bibliotecas interessantes

Já são do Python (*built-in*):

- `datetime`
- `decimal`
- `fractions`
- `math`
- `os`
- `random`

Precisa instalar (via `pip`):

- `numpy`
- `pandas`
- `matplotlib`

Entre muitas outras!

Como utilizar uma biblioteca — `import`

Comando: `import bib`

- Permite usar a biblioteca inteira, mas precisa colocar o nome antes da função/classe/constante

Ex:

```
1 import math
2
3 print(math.sin(2.3), math.pi)
```

Como utilizar uma biblioteca — `import`

Comando: `import bib as outro_nome`

- Permite usar a biblioteca inteira, mas precisa colocar o `outro_nome` antes da função/classe/constante

Ex:

```
1 import math as m
2
3 print(m.sin(2.3), m.pi)
```

Usando para bibliotecas com nomes "grandes"

- Ex: `import numpy as np`

Como utilizar uma biblioteca — `import`

Comando: `from bib import algo`

- permite usar `algo` sem colocar o `bib` antes

Ex:

```
1 from math import sin, pi
2
3 print(sin(2.3), pi)
```

Como utilizar uma biblioteca — `import`

Comando: `from biblioteca import *`

- permite usar toda a biblioteca sem colocar o `bib` antes

Ex:

```
1 from math import *
2
3 print(sin(2.3), pi)
```

Deve-se evitar o uso, pois o código fica menos legível

- E podem haver conflitos entre os nomes
- Você não sabe quais nomes foram importados

Modularização

Você pode fazer `import` de um arquivo seu!

- Você pode ter vários arquivos na mesma pasta
- E fazer `import arquivo` para importar o arquivo.py

paridade.py

```
1 def par(n):
2     return n % 2 == 0
3
4 def impar(n):
5     return not par(n)
```

prog.py

```
1 from paridade import par
2
3 n = int(input("Entre com n: "))
4
5 if par(n):
6     print(n, "é par")
7 else:
8     print(n, "é impar")
```

Um cuidado

Quando um arquivo é importado, ele é executado!

- Ou seja, se você tiver comandos nele, esses comandos são executados também!

Exemplo (arquivo `lista.py`):

```
1  def le_lista(n):
2      lista = []
3      for i in range(n):
4          lista.append(float(input()))
5      return lista
6
7  def soma_valores(lista):
8      soma = 0
9      for x in lista:
10         soma += x
11     return soma
12
13 n = int(input())
14 lista = le_lista(n)
15 print(soma_valores(lista))
```

Ao fazer `import lista.py`, as linhas 13, 14 e 15 serão executadas

Solução

Uma solução é escrever o código da seguinte forma:

```
1  def le_lista(n):
2      lista = []
3      for i in range(n):
4          lista.append(float(input()))
5      return lista
6
7  def soma_valores(lista):
8      soma = 0
9      for x in lista:
10         soma += x
11     return soma
12
13 if __name__ == "__main__":
14     n = int(input())
15     lista = le_lista(n)
16     print(soma_valores(lista))
```

`__name__` guarda o nome do módulo atual

- E é igual a `"__main__"` se o arquivo não foi importado
- Isto é, ele é o arquivo inicialmente executado pelo Python

Solução melhor

```
1  def le_lista(n):
2      lista = []
3      for i in range(n):
4          lista.append(float(input()))
5      return lista
6
7  def soma_valores(lista):
8      soma = 0
9      for x in lista:
10         soma += x
11     return soma
12
13 def main():
14     n = int(input())
15     lista = le_lista(n)
16     print(soma_valores(lista))
17
18 if __name__ == "__main__":
19     main()
```

Agora você pode executar os comandos quando quiser!

- E a ordem de definição das funções não importa
- Desde que o `if __name__ == "__main__"` esteja no final

Modularização

Utilizando vários arquivos, podemos:

- Organizar melhor o nosso código
- Dividir responsabilidades entre os arquivos
  - Um módulo poderia cuidar do processamento dos dados
  - Outro módulo poderia cuidar da exibição dos resultados
  - etc.
- Compartilhar todo/parte do nosso código
- Colaborar com outros membros da equipe

Conceitos avançados

Uma variável pode guardar uma função

- E podemos chamar a função que a variável guarda...

Ex:

```
1  def data_br(dia, mes, ano):
2      return str(dia) + "/" + str(mes) + "/" + str(ano)
3
4  def data_us(dia, mes, ano):
5      return str(mes) + "/" + str(dia) + "/" + str(ano)
6
7  def data_iso(dia, mes, ano):
8      return str(ano) + "-" + str(mes) + "-" + str(dia)
9
10 formato = data_br
11 print(formato(31, 12, 2020))
12 formato = data_us
13 print(formato(31, 12, 2020))
14 formato = data_iso
15 print(formato(31, 12, 2020))
```

Conceitos avançados

E, com isso, uma função pode receber uma função como parâmetro!

```
1  import paridade
2
3  def seleciona(lista, funcao):
4      nova_lista = []
5      for x in lista:
6          if funcao(x):
7              nova_lista.append(x)
8      return nova_lista
9
10 lista = [1, 2, 3, 4, 5, 6, 7, 8]
11
12 pares = seleciona(lista, paridade.par)
13 print(pares)
14 impares = seleciona(lista, paridade.impar)
15 print(impares)
```

lambda

Com o comando lambda você pode criar uma função anônima

```
1  def seleciona(lista, funcao):
2      nova_lista = []
3      for x in lista:
4          if funcao(x):
5              nova_lista.append(x)
6      return nova_lista
7
8  lista = [1, 2, 3, 4, 5, 6, 7, 8]
9
10 pares = seleciona(lista, lambda x: x % 2 == 0)
11 print(pares)
12 impares = seleciona(lista, lambda x: x % 2 != 0)
13 print(impares)
```

Sintaxe:

- lambda <parâmetros>: <expressão>
- <parâmetros> é zero ou mais parâmetros da função
- <expressão> é uma única linha de código a ser executada
- O valor devolvido pela função é o valor da <expressão>

Exercício

Use a função `seleciona` e a biblioteca `math` para criar uma função que dada uma lista de números reais, devolve uma lista dos números `x` tal que `x` em radianos está no primeiro quadrante.

Exercício

a) Faça uma função `mapeia` que, dada uma lista `l` e uma função de um parâmetro `f`, devolve uma nova lista onde `f` foi aplicada a cada elemento de `l`.
   - Por exemplo, se `l = [1, 2, 3]` e `f(x) = x * x`, então a nova lista é `[1, 4, 9]`

b) Use a função que você criou para, dada uma lista, encontrar uma nova lista com todos os seus elementos elevado ao quadrado.

Exercício

a) Faça uma função `combina` que permite aplicar uma função `f` sobre uma lista `l` para combinar os seus resultados e obter um único valor.
   - Ex: Podemos somar todos os elementos de uma lista de números
   - Ex: Podemos multiplicar todos os elementos de uma lista de números
   - Ex: Podemos fazer `and` de vários valores booleanos

b) Use a função que você criou para concatenar a representação decimal de uma lista de números inteiros positivos
   - Ex: se a lista é `[1, 2, 0, 15]`, o resultado é `'12015'`