

FUNÇÕES: EXCEÇÕES, MODULARIZAÇÃO E MAIS

Santiago Valdés Ravelo
[https://ic.unicamp.br/~santiago/
ravelo@unicamp.br](https://ic.unicamp.br/~santiago/ravelo@unicamp.br)

MC102 - Algoritmos e
Programação de
Computadores

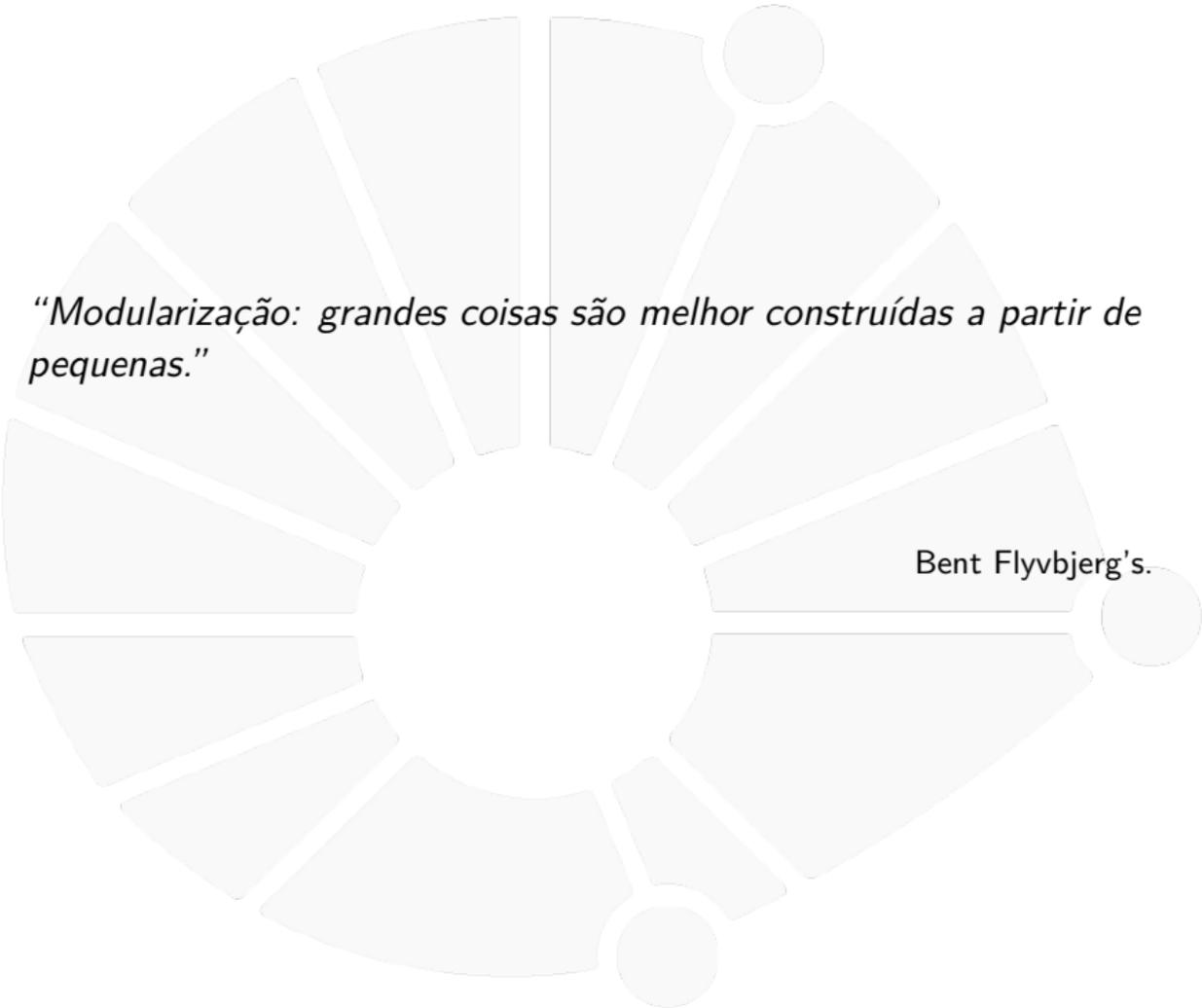
04/25

11



UNICAMP



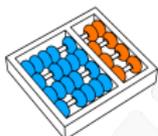


“Modularização: grandes coisas são melhor construídas a partir de pequenas.”

Bent Flyvbjerg's.



DÚVIDAS DA AULA ANTERIOR

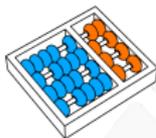


Dúvidas selecionadas

- ▶ Posso definir o valor padrão de um parâmetro como o valor de outro parâmetro? Por exemplo: `def f(x, y = x)`
- ▶ É recomendável usar uma função sem input e output para executar uma lista de passos repetitiva?
- ▶ Existe algum motivo para quando passar uma lista como argumento pra uma função, ela alterar a lista ao invés de fazer uma copia? Porque acredito que isso seja uma escolha na hora de criar a linguagem.
- ▶ Não entendi como acontece a atribuição de um valor padrão para um parâmetro.
- ▶ Mesmo com a diferenciação entre as variáveis locais e globais, é preferível usar variáveis diferentes?
- ▶ Seria certo dizer que "funções" retornam um valor, enquanto métodos não retornam nada?
- ▶ Se eu quiser reutilizar uma função feita em outro programa, há uma forma de importar ela para um novo arquivo sem precisar copiar?
- ▶ O parâmetro de uma função cria uma cópia do objeto de entrada para manipulá-la ou manipula diretamente o próprio objeto? Porque, se o objeto é uma lista e o parâmetro for uma cópia dessa lista, não vejo por que a lista original é modificada dentro da função. Também não entendi o que é o escopo `built in`.
- ▶ Diferença entre objeto e classe.
- ▶ Eu posso fazer uma função que retorna outra função?
- ▶ Para usar uma função dentro de outra função, eu devo definir a segunda função dentro a primeira ou posso definir fora?
- ▶ Não entendi as variáveis `enclosing`.
- ▶ Por que uma função seria definida dentro de outra? Por que não a definir globalmente?



EXCEÇÕES



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)

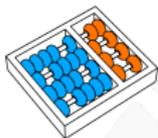
```

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

```



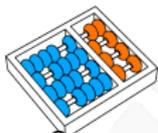
Pilha de Chamadas

```
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**:

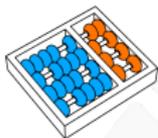
- ▶ O seu programa termina assim que acontece a exceção.
- ▶ Com o Python imprimindo a pilha de execução.
- ▶ Mas nem sempre é isso que queremos...

```
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!

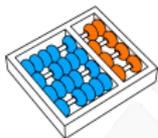


try...except

- ▶ Vamos tentar executar o código (**try**):
 - ▶ Um bloco de código onde pode aparecer uma exceção.
- ▶ E lidamos 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     n = le_numero("Entre com n: ")
10    if n % 2 == 0:
11        print(n, "é par")
12    else:
13        print(n, "é impar")
```



Observações sobre `try`

Você pode capturar várias exceções no mesmo `except`:

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

Você pode ter vários `excepts`:

```
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:

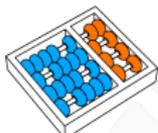
- ▶ o que pode ocultar bugs no seu código, cuidado!

Após todos os `excepts`, você pode ter um `else`.

- ▶ O que fazer se nenhuma exceção ocorrer?

Após todos os `excepts`, você pode ter um `finally`.

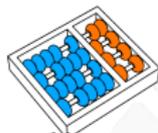
- ▶ Executado independentemente de ter exceção.
- ▶ Sempre é a última coisa a ser feita.



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 def le_lista_de_positivos(n):
7     lista = []
8     for i in range(n):
9         lista[i] = le_numero_positivo()
10    return lista
11
12 try:
13     lista = le_lista_de_positivos(5)
14     print(lista)
15 except ValueError as e:
16     print(e)
```

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



BIBLIOTECAS E MODULARIZAÇÃO



Bibliotecas

O conceito de bibliotecas de código é muito importante:

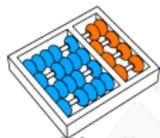
- ▶ A ideia é compartilhar código já escrito.
- ▶ De uma maneira organizada e documentada.
- ▶ Para que outros programadores possam reutilizar.

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:

- ▶ Não precisa reinventar a roda!



Algumas bibliotecas interessantes

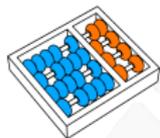
Já são do Python (*built-in*):

- ▶ **datetime**
- ▶ **decimal**
- ▶ **fractions**
- ▶ **math**
- ▶ **os**
- ▶ **random**

Precisa instalar (via **pip**):

- ▶ **numpy**
- ▶ **pandas**
- ▶ **matplotlib**

Entre muitas outras!



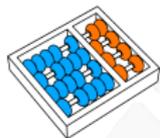
Como utilizar uma biblioteca — `import`

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`:

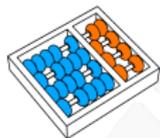
- ▶ Permite usar a biblioteca inteira, mas precisa colocar o `outro_nome` antes da função/classe/constante.

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`.



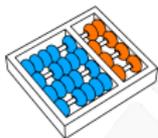
Como utilizar uma biblioteca — `import`

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)
4
```



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:

- ▶ 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.



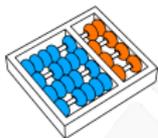
Solução

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.

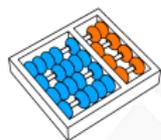


Solução melhor

```
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.



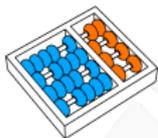
Modularização

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.



FUNÇÕES COMO VARÁVEIS E PARÂMETROS



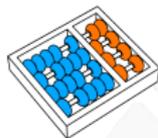
Conceitos avançados

Uma variável pode guardar uma função:

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

Ex:

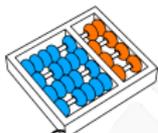
```
1 def data_br(dia, mes, ano):
2     return str(dia) + "/" + str(mes) + "/" + str(ano)
3
4 def data_us(dia, mes, ano):
5     return str(mes) + "/" + str(dia) + "/" + str(ano)
6
7 def data_iso(dia, mes, ano):
8     return str(ano) + "-" + str(mes) + "-" + str(dia)
9
10 formato = data_br
11 print(formato(31,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



Conceitos avançados

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:

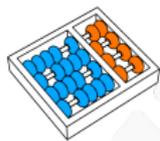
```
1 def seleciona(lista, funcao):
2     nova_lista = []
3     for x in lista:
4         if funcao(x):
5             nova_lista.append(x)
6     return nova_lista
7
8 lista = [1, 2, 3, 4, 5, 6, 7, 8]
9
10 pares = seleciona(lista, lambda x: x % 2 == 0)
11 print(pares)
12 impares = seleciona(lista, lambda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



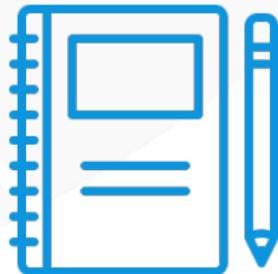
EXERCÍCIOS

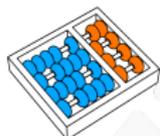


Listas e repetição



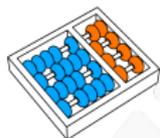
Vamos fazer alguns exercícios?





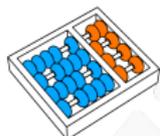
Exercício 1

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 2

- 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 3

- 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'**.

FUNÇÕES: EXCEÇÕES, MODULARIZAÇÃO E MAIS

Santiago Valdés Ravelo
[https://ic.unicamp.br/~santiago/
ravelo@unicamp.br](https://ic.unicamp.br/~santiago/ravelo@unicamp.br)

MC102 - Algoritmos e
Programação de
Computadores

04/25

11



UNICAMP

