Programação I
Aula 17 — Correção de programas

Pedro Vasconcelos
DCC/FCUP

Nesta aula

Classes de erros

- Erros em programas:
  - erros sintáticos;
  - erros de execução (exceções);
  - erros lógicos (resposta incorreta ou não termina)
- São responsabilidade do programador: o computador tem sempre razão!
- Todos os programadores devem aprender a detectar e corrigir erros
- É importante aprender a corrigir erros de forma constructiva.
- Não devem tentar modificações ao acaso!

Erros sintáticos

```
  File "fich.py", line 15
    ....
SyntaxError: invalid syntax
```

- A mensagem do interpretador inclui o *número da linha* (o IDLE assinala a vermelho)
- Mas a *causa* pode estar em linhas anteriores
- Alguns erros sintáticos comuns:
  - esquecer de fechar parêntesis ou aspas
  - usar uma palavra reservada como variável
  - esquecer dois-pontos (`:`) no início de um bloco
  - usar = em vez de ==
  - usar indentação inconsistente

Erros de execução

```
>>> palindromo("abba")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "palindromo.py", line 7, in palindromo
    if txt[i] != txt[j]:
IndexError: string index out of range
```

- A exceção indica a causa imediata do erro

  IndexError: um índice inválido numa sequência
- O “traceback” indica a sequência de instruções que antecedeu a exceção

Erros lógicos

```
>>> palindromo("abba")
False
```

- Resposta incorreta ou um programa que não termina
- A causa pode estar:
  - na *compreensão* do problema;
  - no *algoritmo* para resolução;
  - na *implementação* em Python.
- Mais difícil saber por onde começar!

Corrigir erros lógicos

1. Leia com atenção o enunciado do problema
2. Escreva no papel alguns casos de teste simples
3. Leia o enunciado mais uma vez!
4. Se necessário, decomponha o problema em funções
5. Cada função deve ter um objectivo bem definido
6. Não espere pelo fim: teste cada função independente no interpretador

Execução passo-a-passo

`http://pythontutor.com`

- Útil para compreender conceitos fundamentais
- Também pode ajudar a compreender o funcionamento de um programa
- Desvantagens:
  - difícil ou impossível seguir para programas grandes
  - não ilustra as propriedades do *algoritmo*
  - não mostra *porquê* um programa (não) funciona
- É melhor usar apenas com exemplos pequenos

Exemplo: calcular potências

```
def potencia(x: int, n: int) -> int:
    r = 1
    while n > 1:
        if n%2 != 0:
            r = r*x
        x = x*x
        n = n//2
    return r

print(potencia(2,3))
```

- A função acima tem um erro
- Tente executar passo-a-passo para corrigir
- Compare com a solução desenvolvida na aula 16

Doctests

- Documentação de resultados esperados de funções
- Incluidos na documentação das funções
- Sintaxe baseada na interação com o interpretador
- Podem ser testados automaticamente
- Usados no sistema *Codex* para as submissões de exercícios

Exemplo: calcular raizes

- Definir uma função

  `raiz(x: int) -> int`

  que calcula a *parte inteira da raiz quadrada*:

$$\begin{aligned}\lfloor\sqrt{2}\rfloor &= \lfloor 1.414\ldots\rfloor &= 1\\ \lfloor\sqrt{4}\rfloor &= \lfloor 2\rfloor &= 2\\ \lfloor\sqrt{7}\rfloor &= \lfloor 2.645\ldots\rfloor &= 2\\ \lfloor\sqrt{9}\rfloor &= \lfloor 3\rfloor &= 3\end{aligned}$$

- Sem usar operações sobre *floats* nem o módulo *math*

Exemplo: calcular raizes (cont.)

```
def raiz(x: int) -> int:
    '''Parte inteira da raiz quadrada; exemplos:
    >>> raiz(2)
    1
    >>> raiz(4)
    2
    >>> raiz(7)
    2
    >>> raiz(9)
    3
    '''
    k = 0
    while k*k < x:
        k = k+1
    return k
```

Testar

```
>>> import doctest
>>> doctest.testmod()
```

- A função *testmod()* procura testes da forma "`>>> ...`" nas *docstrings*
- Executa e compara resultados obtidos com os valores documentados
  - se forem iguais: o teste passa
  - se forem diferentes: assinala um erro

Quando os testes falham

```
File "raiz.py", line 14, in __main__.raiz
Failed example:
    raiz(2)
Expected:
    1
Got:
    2
**********************************************************************
File "raiz.py", line 16, in __main__.raiz
Failed example:
    raiz(7)
Expected:
    2
Got:
    3
```

Qual a causa do erro?

- Falham as raizes de números que não são quadrados perfeitos:

```
Failed example:
    raiz(2)
Expected:
    1
Got:
    2
```

- Devemos aproximar por *defeito* e não por *excesso*
- Podemos testar esses casos no final do ciclo

Versão corrigida

```
def raiz(x: int) -> int:
    :
    k = 0
    while k*k < x:
        k = k+1
    # é quadrado perfeito?
    if k*k == x:
       return k    # raiz exata
    else:
       return k-1 # aproximação por defeito
```

Asserções

```
assert condição, mensagem
```

Declara uma condição que se deve verificar num determinado ponto do programa.

Se a condição for True: a execução continua

Se a condição for False: aborta com uma exceção `AssertionError`

Como usar asserções

- Escrever condições de funcionamento correto do programa
- Permitem detetar erros de programação mais cedo
- Vamos ver um caso de uso: pré-condições à entrada de funções

Pré-condições

- Condições que se devem verificar à entrada de uma função
- Uma função deve calcular corretamente *apenas* quando as pré-condições se verificam
- A resposabilidade de garantir as pré-condições é de quem *chama a função*

Exemplo: calcular o binomial (exercício 5.4)

- Definir uma função `binom(n: int, k: int) -> int` para calcular coeficientes binomiais:

$$\binom{n}{k} = \frac{n!}{k! \times (n-k)!}$$

- Correspondem aos valores do *triângulo de Pascal*:

$$\begin{array}{ccccccc} & & & 1 & & & (n=0) \\ & & 1 & & 1 & & (n=1) \\ & 1 & & 2 & & 1 & (n=2) \\ 1 & & 3 & & 3 & & 1 \quad (n=3) \\ & \vdots & & & & \vdots & \end{array}$$

- Pré-condições: $k \geq 0 \wedge k \leq n$

Em Python

```
def binom(n: int, k: int) -> int:
    "Calcular binomial de n, k."
    assert k>=0 and k<=n, "pré-condições de binom"
    return fact(n) // (fact(k) * fact(n-k))

def fact(n: int) -> int:
    "Calcular o factorial de n."
    ⋮
```


Testando

```
>>> binom(1,3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "binom.py", line 11, in binom
    assert k>=0 and k<=n, "pré-condições de binom"
AssertionError: pré-condições de binom
```

- A função `binom` está correta; este exemplo viola as pré-condições
- O uso da asserção deteta o erro imediatamente
- Evita usar um resultado inválido noutra parte do programa

Comparações

Cuidado com as comparações com três operandos.

```
# programa original
if not (a<b<c):
    ⋮
```

```
# errado!
if a>=b>=c:
    ⋮
```

```
# correcto
if a>=b or b>=c:
    ⋮
```

Para não se enganar é preferivel escrever sempre comparações binárias (`a<b and b<c`).

Limites de ciclos

Seja `xs` uma lista de inteiros:

```
for i in range(len(xs)):
    if xs[i]%2==0:
        del xs[i]
```

Qual é o erro?

- o valores do `range` não são re-calculados dentro do ciclo
- logo, ao removeremos elementos vamos usar índices inválidos (IndexError)

Ciclos com `else`

Programa A

```
i = 0
while i<100:
    i = i+1
    if i%2 == 0:
        continue
    else:
        print(i)
```

Programa B

```
i = 0
while i<100:
    i = i+1
    if i%2 == 0:
        continue
else:
    print(i)
```

- O programa B terá um erro sintático?
- Não: `else` também pode ser usado com ciclos `for`/`while` (não abordamos na aulas de Programação I)
- Mas não é equivalente ao programa A
- Cuidado com a indentação!