Nesta aula...

Conteúdo

1 Detecção e correcção de erros **1**

1.1 Erros sintáticos 1

1.2 Erros de execução 2

1.3 Erros semânticos 5

1.4 Erros semânticos 6

1 Detecção e correcção de erros

Detecção e correcção de erros

- diferentes tipos de erros :
  - o programa é rejeitado pelo interpretador
  - é aceite, mas dá uma mensagem de erro durante a execução
  - não dá mensagens de erro, mas não faz o que eu quero!
- são causados pelo programador: o computador tem sempre razão!
- todos os programadores devem aprender a detectar e corrigir erros
- não devem tentar modificações ao acaso!
- é importante desenvolver um *método* construtivo

Tipos de erros

Sintácticos: o programa é rejeitado pelo interpretador Python

Execução: dá uma mensagem de erro durante a execução

Semântico: o programa não faz o que eu quero!

1.1 Erros sintáticos

Erros sintáticos

Erros sintáticos

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

- a mensagem indica a *linha* onde o erro é *detectado*
- a *causa* do erro pode estar nas linhas anteriores
- alguns erros sintáticos comuns:
  - indentações erradas; por exemplo um "espaço" antes de um `def`.
  - esquecer de fechar parêntesis ou aspas
  - usar uma palavra reservada como variável
  - esquecer dois-pontos (`:`) no início de um bloco
  - usar = em vez de ==

Corrigir erros sintáticos

- observar atentamente a linha onde o erro foi detectado
- procurar desalinhamento na indentação dos blocos
- procurar parêntesis ou aspas abertos nas linhas anterioes
- verificar a *estrutura* de blocos `if-then-else`, `while`, `for`
- no editor Emacs, IDLE, Eric...: as *cores* indicam palavras chave, cadeias de caracteres, etc. Trabalhe sempre num ficheiro com a extensão `.py`.

1.2 Erros de execução

Erros de execução

Erros de execução

Excepções: o programa termina com uma mensagem de erro

Ciclo infinito: o programa não termina!

Exemplo: verificar palíndromos

```
def palindromo(str):
  "Verifica se uma cadeia é igual ao seu reverso."
  i = 0
  j = len(str)
  while i<j:
      if str[i]!=str[j]:
          return False
      i = i+1
      j = j-1
  return True
```

Erros de execução

```
>>> palindromo("olamundo")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "/tmp/python-18422-ah.py", line 6,
in palindromo
IndexError: string index out of range
```

Excepção: diz qual o erro (`IndexError`: uso de um índice inválido)

Traceback: diz onde foi detectado (linha 6 da função `palindromo`)

Corrigir erros de execução

```
def palindromo(str):
   ⋮

   if str[i]!=str[j]:      # linha 6
   ⋮
```

- identificar o ponto onde erro foi detectado
- acrescentar código para verificar valores de variáveis

Corrigir erros de execução (2)

Executando o programa novamente:

```
>>> palindromo("olamundo")
i= 0 j= 8 len= 8
Traceback (most recent call last):
  ...
IndexError: string index out of range
```

Índices válidos:

$$0 \leq i,\, j \leq len - 1$$

Temos:

$$i = 0,\, j = 8,\, len = 8$$

Logo: o valor de j está errado!

Corrigir erros de execução (3)

```
def palindromo(str):
  "Verifica se uma cadeia é igual ao seu reverso."
  i = 0
  j = len(str)   # erro: j>len-1
  while i<j:
    if str[i]!=str[j]:
       return False
    i = i+1
    j = j-1
  return True
```

Asserções

$$assert\ cond,\ msg$$

- se a condição for verdadeira, a asserção não faz nada
- se a condição for falsa, lança uma excepção `AssertionError: msg`
- ideia: marcar condições que devem ser verdadeiras se o programa estiver correcto

Acrescentar asserções

```
def palindromo(str):
  "Verifica se uma cadeia é igual ao seu reverso."
  i = 0
  j = len(str)    # erro
  while i<j:
    assert 0<=i<len(str), "limites do índice i"
    assert 0<=j<len(str), "limites do índice j"
    if str[i]!=str[j]:
        return False
    i = i+1
    j = j-1
  return True
```

Execução com asserções

```
>>> palindromo("olamundo")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "/tmp/python-9298rhy.py",
line 8, in palindromo
AssertionError: limites do índice j
```

Quando a execução não termina

- Problemas de programação elementares são resolvidos rapidamente num computador actual (tipicamente: menos de 1 segundo)
- Se o programa demora muito tempo, provavelmente tem um *ciclo infinito*
- Recursão sem fim: programa termina com um erro `Maximum recursion depth exceeded`

Evitar ciclos infinitos

```
while x>0 and y<0:
   # modifica x, y
   ⋮
```

- para que o ciclo termine, a condição deve ser `False` em algum momento
- acrescentar código para verificar se `x` e `y` estão a ser modificados correctamente

Evitar ciclos infinitos (2)

- verificar se as variáveis na condição `while` são alteradas no corpo do ciclo
- ter presente quais os *intervalos* desejados para índices
- se possível: escrever um ciclo simples em vez de dois ciclos embricados
- se possível: usar ciclos `for` (terminam sempre!)

Evitar recursão infinita

```
def fact(n):
   "Calcula o factorial recursivamente."
   if n==0:
     return 1
   else:
     return n*fact(n-1)
```

- verificar se algum argumento na chamada recursiva fica menor
- verificar o caso base está bem definido (não recursivamente)

1.3 Erros semânticos

Erros semânticos

Erros semânticos

- o programa dá um resultado, mas não é correcto!
- o computador não advinha o que eu quero
- logo: o programa que escrevi não resolve o problema pretendido

Os erros semânticos são os mais difíceis de corrigir

Corrigir erros semânticos

Re-pensar o problema com papel e lápis:

- *decompor* o problema em funções pequenas
- cada função deve ser um *objectivo* bem definido
- pensar qual deverá ser o *resultado* de cada função com valores pequenos
- *testar* as funções no interpretador com esses valores
- o erro pode estar numa *solução mal compreendida* e não no programa
- o erro pode estar numa *especificação mal compreendida* e não no programa

1.4 Erros semânticos

Testes

Sobre os testes

- As funções – que devem ser simples, claras e curtas – podem ser automaticamente sujeitas a um conjunto de testes, usando o módulo `doctest`.
- O número de dados possíveis de uma função é (praticamente) infinito. Por isso os testes podem servir para mostrar que uma função está errada, mas nunca que está correcta.

Relembrando o `doctest`...exemplo, parte I

```
def factorial(n):
    """Retorna
         - o factorial de n se n e' um inteiro
           nao negativo
         - 'False' caso contrario
    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial("piu")
    False
    """
    if type(n)!=int or n<0:
        return False
```

(continua)

Relembrando o `doctest`...exemplo, parte I

```
def factorial(n):
    ....
    r=1
    factor = 2
    while factor <= n:
        r = r*factor
        factor = factor+1
    return r

def _test():
    import doctest
    doctest.testmod()

if __name__ == "__main__":
    _test()
```