Introdução à Ciência da Computação

Disciplina: 113913

Prof. Edison Ishikawa

python™

Python 3.0

Capítulo 6

Iteração

UnB | CIC

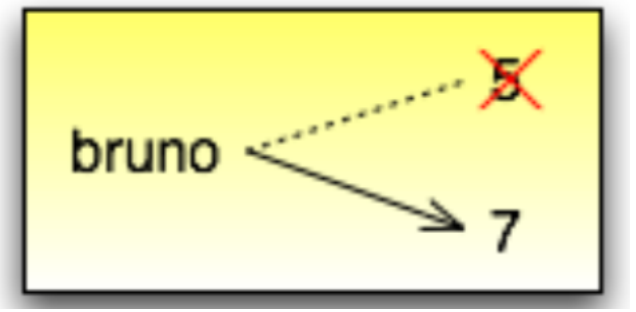
Sumário

- Reatribuições
- O comando while
- Tabelas
- Tabelas de duas dimensões
- Encapsulamento e generalização
- Mais encapsulamento
- Variáveis locais
- Mais generalização
- Funções

Reatribuições

- É permitido fazer uma mais de uma atribuição para uma mesma variável;
- Assim, a novo atribuição faz uma variável existente referir-se a um **novo valor** (sem se referir mais ao antigo).

```
bruno = 5
print(bruno, end=" ")
bruno = 7
print(bruno)
```



Atribuição vs. Igualdade

- Como Python usa o sinal de igual (=) para atribuição, existe a tendência de lermos um comando como a = b como um comando de igualdade. Mas não é!
- Em primeiro lugar, a igualdade é comutativa (a= 7 e 7 = a), mas a atribuição não é (o comando a = 7 é permitido, mas 7 = a não é).
- Lembrando de reatribuição:

```
a = 5
b = a # a e b agora são iguais
b = 3 # a e b não são mais iguais
```

O comando while

- O comando ***while*** é utilização para automatização de tarefas repetitivas.
- A repetição também pode ser denominada de **iteração**
- Exemplo: função contagemRegressiva com o comando while:

```
def contagemRegressiva(n):
        while n>0:
                print(n)
                n = n - 1
        print("Fogo!")
```

  - O comando while significa: **Enquanto (while) n > 0, siga exibindo o valor de n e diminuindo 1 do valor de n. Quando chegar a 0, imprima a palavra Fogo!"**

O comando while

- Fluxo de execução para o comando while:
  1. Teste a condição, resultando 0 ou 1.
  2. Se a condição for falsa (0), saia do comando while e continue a execução a partir do próximo comando.
  3. Se a condição for verdadeira (1), execute cada um dos comandos.

- O corpo consiste em todos os comandos abaixo do cabeçalho, com a mesma indentação.
- Este tipo de fluxo é chamado de **loop (ou laço)** porque o terceiro passo cria um loop de volta ao topo.

Tabelas

- Loops podem ser utilizados para gerar dados tabulares.
- A **string '\t'** representa um caracter de tabulação. Este caracter desloca o curso para direita até que ele encontre uma das marcas de tabulação.

```
import math
x = 1
while x < 10.0:
        print(x, '\t', math.log(x)/math.log(2.0))
        x = x + 1
```

```
1       0.0
2       1.0
3       1.5849625007211563
4       2.0
5       2.321928094887362
6       2.584962500721156
7       2.807354922057604
8       3.0
9       3.1699250014423126
```

Tabelas de duas dimensões

- Em uma tabela de duas dimensões, o valor desejado se encontra na intersecção entre uma linha e uma coluna. Exemplo: tabela de multiplicação.
- Podemos iniciar uma tabela de multiplicação com um loop que imprima os múltiplos de 2, todos em uma linha:

```
i = 1
while i <= 6:
        print (2*i,'\t', end="")
        i = i + 1
print()
```

```
2	4	6	8	10	12
```

- Próximo passo: **encapsular e generalizar**

Encapsulamento e Generalização

- **Encapsulamento** é o processo de wrapping de um pedaço de código em uma função.
- **Generalização** significa tomar algo que é específico, tal como imprimir múltiplos de 2, e torná-lo mais geral, como imprimir os múltiplos de qualquer inteiro.
- A função abaixo encapsula o loop anterior e generaliza-o para imprimir múltiplos de n:

```
def imprimeMultiplos(n):
        i = 1
        while i <= 6:
                print(n*i, '\t', end="")
                i = i + 1
        print()
```

```
3       6       9       12      15      18
4       8       12      16      20      24
```

Encapsulamento e Generalização

- Podemos imprimir a tabela de multiplicação chamando a função imprimeMultiplos repetidamente com argumentos diferentes.

```
i = 1
while i <= 6:
        imprimeMultiplos(i)
        i = i+1
```

```
1       2       3       4       5       6
2       4       6       8       10      12
3       6       9       12      15      18
4       8       12      16      20      24
5       10      15      20      25      30
6       12      18      24      30      36
```

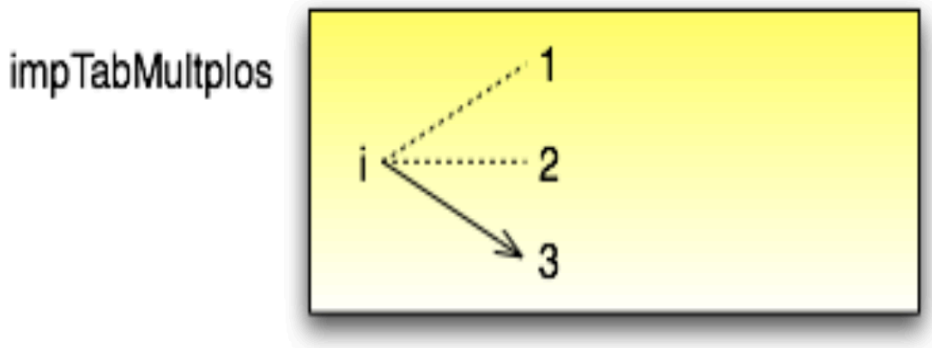
Mais encapsulamento

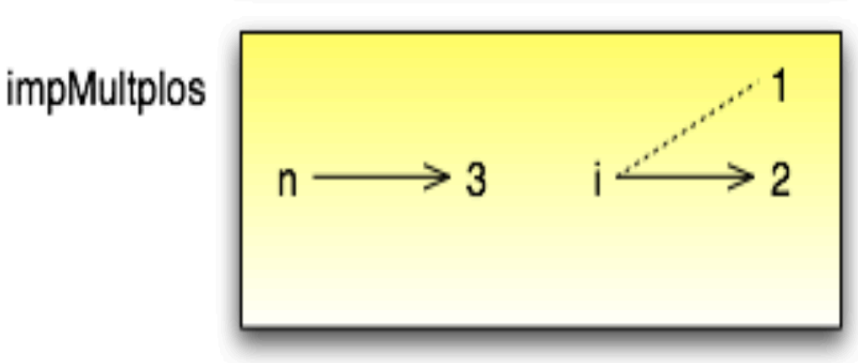
- Podemos pegar o código anterior e encapsular numa função:

```
def imprimeMultiplos(n):
        i = 1
        while i <= 6:
                print(n*i, '\t', end="")
                i = i + 1
        print()

def imprimeTabMult():
        i = 1
        while i <= 6:
                imprimeMultiplos(i)
                i = i + 1

imprimeTabMult()
```

Variáveis Locais

- Variáveis criadas dentro de uma função são locais, ou seja, você não pode acessar uma variável local de fora da função em que ela foi definida.
- Isto significa que podemos ter múltiplas variáveis com o mesmo nome desde que elas não estejam dentro da mesma função.





Mais generalização

- Quando generalizamos uma função apropriadamente, podemos obter um programa com capacidades não planejadas.
- Podemos modificar a funcão imprimeTabMult para:
  - ✓ Imprimir múltiplos de 7, 8, 9, …
  - ✓ Imprimir apenas metade da tabela já que temos valores repetidos

Funções

- Vantagens da utilização de funções:

1. Dar um nome para uma sequência de comandos torna o programa mais fácil de ler e depurar.
2. Dividir um programa longo em funções permite que você separe partes do programa, depure-as isoladamente, e então as componha como um todo.
3. Funções facilitam tanto recursão quanto iteração
4. Funcões bem projetadas são frequentemente úteis para muitos programas.

Referências

[1] Aprenda Computação com Python 3.0, Versão 1. Allen Downey, Jeff Elkner and Chris Meyers