

BCM0505-22 – Processamento da Informação

Funções

Carla Negri Lintzmayer
carla.negri@ufabc.edu.br
<http://professor.ufabc.edu.br/~carla.negri/>

Outline

Funções

Chamando funções

Definindo funções

Escopo de variáveis

Fluxo de execução

Resumo

Exemplo

Pratique!

Funções

Funções

- São construções de código que nos permitem transferir o controle do fluxo de execução para diferentes partes do código.

Funções

- São construções de código que nos permitem transferir o controle do fluxo de execução para diferentes partes do código.
 - Basicamente, agrupam uma sequência de comandos que serão executados quando a utilizarmos.

Funções

- São construções de código que nos permitem transferir o controle do fluxo de execução para diferentes partes do código.
 - Basicamente, agrupam uma sequência de comandos que serão executados quando a utilizarmos.
- São úteis para separar diferentes tarefas dentro de um mesmo programa.

Funções

- São construções de código que nos permitem transferir o controle do fluxo de execução para diferentes partes do código.
 - Basicamente, agrupam uma sequência de comandos que serão executados quando a utilizarmos.
- São úteis para separar diferentes tarefas dentro de um mesmo programa.
 - Podemos dividir a solução de um problema na solução de problemas menores, que podem ser resolvidos de forma independente.

Funções

- São construções de código que nos permitem transferir o controle do fluxo de execução para diferentes partes do código.
 - Basicamente, agrupam uma sequência de comandos que serão executados quando a utilizarmos.
- São úteis para separar diferentes tarefas dentro de um mesmo programa.
 - Podemos dividir a solução de um problema na solução de problemas menores, que podem ser resolvidos de forma independente.
- São úteis para reutilizar código facilmente.

Funções

- São construções de código que nos permitem transferir o controle do fluxo de execução para diferentes partes do código.
 - Basicamente, agrupam uma sequência de comandos que serão executados quando a utilizarmos.
- São úteis para separar diferentes tarefas dentro de um mesmo programa.
 - Podemos dividir a solução de um problema na solução de problemas menores, que podem ser resolvidos de forma independente.
- São úteis para reutilizar código facilmente.
- Já vimos e utilizamos algumas funções:

Funções

- São construções de código que nos permitem transferir o controle do fluxo de execução para diferentes partes do código.
 - Basicamente, agrupam uma sequência de comandos que serão executados quando a utilizarmos.
- São úteis para separar diferentes tarefas dentro de um mesmo programa.
 - Podemos dividir a solução de um problema na solução de problemas menores, que podem ser resolvidos de forma independente.
- São úteis para reutilizar código facilmente.
- Já vimos e utilizamos algumas funções:
 - `input()`, `print()`, `int()`, `float()`, `str()`, etc

Funções

- Para usarmos funções, não é necessário *como* elas funcionam, mas sim *o que* elas fazem.

Funções

- Para usarmos funções, não é necessário *como* elas funcionam, mas sim *o que* elas fazem.
 - Não fazemos ideia de como `input()` lê alguma coisa da entrada padrão, porém sabemos que ela nos devolverá esse valor.

Funções

- Para usarmos funções, não é necessário *como* elas funcionam, mas sim *o que* elas fazem.
 - Não fazemos ideia de como `input()` lê alguma coisa da entrada padrão, porém sabemos que ela nos devolverá esse valor.
 - Não sabemos como `int()` transforma uma string em um inteiro, mas sabemos que é isso que acontecerá.

Funções

- Para usarmos funções, não é necessário *como* elas funcionam, mas sim *o que* elas fazem.
 - Não fazemos ideia de como `input()` lê alguma coisa da entrada padrão, porém sabemos que ela nos devolverá esse valor.
 - Não sabemos como `int()` transforma uma string em um inteiro, mas sabemos que é isso que acontecerá.
- Para isso, funções precisam ser bem ***documentadas***

Funções

- Para usarmos funções, não é necessário *como* elas funcionam, mas sim *o que* elas fazem.
 - Não fazemos ideia de como `input()` lê alguma coisa da entrada padrão, porém sabemos que ela nos devolverá esse valor.
 - Não sabemos como `int()` transforma uma string em um inteiro, mas sabemos que é isso que acontecerá.
- Para isso, funções precisam ser bem ***documentadas***
 - Toda função deve descrever o que é esperado como argumento, e em qual ordem caso haja mais de um.

Funções

- Para usarmos funções, não é necessário *como* elas funcionam, mas sim *o que* elas fazem.
 - Não fazemos ideia de como `input()` lê alguma coisa da entrada padrão, porém sabemos que ela nos devolverá esse valor.
 - Não sabemos como `int()` transforma uma string em um inteiro, mas sabemos que é isso que acontecerá.
- Para isso, funções precisam ser bem ***documentadas***
 - Toda função deve descrever o que é esperado como argumento, e em qual ordem caso haja mais de um.
 - Toda função deve descrever o que será devolvido como resultado final.

Funções

- Toda função tem dois “momentos” importantes

Funções

- Toda função tem dois “momentos” importantes
 - Na **definição**, apenas indicamos qual será o identificador da função (seu nome), quais dados ela deve receber e quais comandos ela deve fazer para atingir seu objetivo.

Funções

- Toda função tem dois “momentos” importantes
 - Na **definição**, apenas indicamos qual será o identificador da função (seu nome), quais dados ela deve receber e quais comandos ela deve fazer para atingir seu objetivo.
 - Na **chamada**, escrevemos o identificador da função que queremos usar e passamos os dados que ela requer.

Funções

- Toda função tem dois “momentos” importantes
 - Na **definição**, apenas indicamos qual será o identificador da função (seu nome), quais dados ela deve receber e quais comandos ela deve fazer para atingir seu objetivo.
 - Na **chamada**, escrevemos o identificador da função que queremos usar e passamos os dados que ela requer.
- A definição de uma função, portanto, não tem efeito durante a execução do código.

Funções

- Toda função tem dois “momentos” importantes
 - Na **definição**, apenas indicamos qual será o identificador da função (seu nome), quais dados ela deve receber e quais comandos ela deve fazer para atingir seu objetivo.
 - Na **chamada**, escrevemos o identificador da função que queremos usar e passamos os dados que ela requer.
- A definição de uma função, portanto, não tem efeito durante a execução do código.
 - Apenas quando ela é chamada é que seus comandos serão executados.

Funções

- Toda função tem dois “momentos” importantes
 - Na **definição**, apenas indicamos qual será o identificador da função (seu nome), quais dados ela deve receber e quais comandos ela deve fazer para atingir seu objetivo.
 - Na **chamada**, escrevemos o identificador da função que queremos usar e passamos os dados que ela requer.
- A definição de uma função, portanto, não tem efeito durante a execução do código.
 - Apenas quando ela é chamada é que seus comandos serão executados.
- Podemos utilizar funções pré-definidas em Python e/ou então construir nossas próprias funções.

Chamando funções

Chamando funções

- O uso de uma função já existente ocorre quando fazemos uma *chamada* para a função.

Chamando funções

- O uso de uma função já existente ocorre quando fazemos uma *chamada* para a função.
- O código para uma chamada a uma função é o nome identificador da função seguido por **argumentos**, que devem estar entre parênteses e separados por vírgulas.

Chamando funções

- O uso de uma função já existente ocorre quando fazemos uma *chamada* para a função.
- O código para uma chamada a uma função é o nome identificador da função seguido por *argumentos*, que devem estar entre parênteses e separados por vírgulas.
 - `nome_da_funcao(argumento1, argumento2, ...)`

Chamando funções

- O uso de uma função já existente ocorre quando fazemos uma *chamada* para a função.
- O código para uma chamada a uma função é o nome identificador da função seguido por *argumentos*, que devem estar entre parênteses e separados por vírgulas.
 - `nome_da_funcao(argumento1, argumento2, ...)`
 - Quantos argumentos e o tipo de cada um depende da definição da função.

Chamando funções

- O uso de uma função já existente ocorre quando fazemos uma *chamada* para a função.
- O código para uma chamada a uma função é o nome identificador da função seguido por *argumentos*, que devem estar entre parênteses e separados por vírgulas.
 - `nome_da_funcao(argumento1, argumento2, ...)`
 - Quantos argumentos e o tipo de cada um depende da definição da função.
- Após chamarmos uma função, ela é executada/avaliada e *devolve* algum valor.

Chamando funções

- O uso de uma função já existente ocorre quando fazemos uma *chamada* para a função.
- O código para uma chamada a uma função é o nome identificador da função seguido por *argumentos*, que devem estar entre parênteses e separados por vírgulas.
 - `nome_da_funcao(argumento1, argumento2, ...)`
 - Quantos argumentos e o tipo de cada um depende da definição da função.
- Após chamarmos uma função, ela é executada/avaliada e *devolve* algum valor.
 - Mesmo que seja nada.

Chamando funções

- O uso de uma função já existente ocorre quando fazemos uma *chamada* para a função.
- O código para uma chamada a uma função é o nome identificador da função seguido por *argumentos*, que devem estar entre parênteses e separados por vírgulas.
 - `nome_da_funcao(argumento1, argumento2, ...)`
 - Quantos argumentos e o tipo de cada um depende da definição da função.
- Após chamarmos uma função, ela é executada/avaliada e *devolve* algum valor.
 - Mesmo que seja nada.
- Assim, uma chamada para uma função também é uma expressão, e por isso pode ser usada da mesma forma como usamos variáveis e literais.

Chamando funções

- O uso de uma função já existente ocorre quando fazemos uma *chamada* para a função.
- O código para uma chamada a uma função é o nome identificador da função seguido por *argumentos*, que devem estar entre parênteses e separados por vírgulas.
 - `nome_da_funcao(argumento1, argumento2, ...)`
 - Quantos argumentos e o tipo de cada um depende da definição da função.
- Após chamarmos uma função, ela é executada/avaliada e *devolve* algum valor.
 - Mesmo que seja nada.
- Assim, uma chamada para uma função também é uma expressão, e por isso pode ser usada da mesma forma como usamos variáveis e literais.
- Um *argumento* pode ser qualquer expressão (variável, literal, outra chamada de função, expressões mais complexas...)

Exemplos

- `input()` devolve uma string, que é o valor lido da entrada padrão, e por isso podemos construir expressões como `int(input())`

```
1 ano = input()
2 ano = int(ano)
```

- `int()` espera receber um único argumento, string ou numérico
 - `int("4")`
 - `int("4"+"5")`
 - `int(23.3)`
 - `int(23**0.5 + 35 - 342 + a)`
- `input()` não precisa receber argumentos para funcionar, apesar de podermos passar uma string que será impressa na saída padrão.

Definindo funções

Definindo funções

- Toda definição de função em Python deve ter o seguinte formato:

```
1 def identificador(parametro1, parametro2, ...):  
2     comandos identados
```

Definindo funções

- Toda definição de função em Python deve ter o seguinte formato:

```
1 def identificador(parametro1, parametro2, ...):  
2     comandos identados
```

- A primeira linha é o *cabeçalho* da função.

Definindo funções

- Toda definição de função em Python deve ter o seguinte formato:

```
1 def identificador(parametro1, parametro2, ...):  
2     comandos identados
```

- A primeira linha é o *cabeçalho* da função.
 - Os *parâmetros* de uma função, que podem não existir, são identificadores de variáveis que poderão ser utilizadas no corpo da função.

Definindo funções

- Toda definição de função em Python deve ter o seguinte formato:

```
1 def identificador(parametro1, parametro2, ...):  
2     comandos identados
```

- A primeira linha é o *cabeçalho* da função.
 - Os *parâmetros* de uma função, que podem não existir, são identificadores de variáveis que poderão ser utilizadas no corpo da função.
 - As linhas seguintes, indentadas em relação ao cabeçalho, são o *corpo* da função.

Definindo funções

- Toda definição de função em Python deve ter o seguinte formato:

```
1 def identificador(parametro1, parametro2, ...):  
2     comandos identados
```

- A primeira linha é o **cabeçalho** da função.
 - Os **parâmetros** de uma função, que podem não existir, são identificadores de variáveis que poderão ser utilizadas no corpo da função.
 - As linhas seguintes, indentadas em relação ao cabeçalho, são o **corpo** da função.
- O corpo de uma função pode conter qualquer comando que já vimos e que vamos ver (inclusive chamadas a outras funções!), exceto outra definição de função.

Definindo funções

- Toda definição de função em Python deve ter o seguinte formato:

```
1 def identificador(parametro1, parametro2, ...):  
2     comandos identados
```

- A primeira linha é o **cabeçalho** da função.
 - Os **parâmetros** de uma função, que podem não existir, são identificadores de variáveis que poderão ser utilizadas no corpo da função.
 - As linhas seguintes, indentadas em relação ao cabeçalho, são o **corpo** da função.
- O corpo de uma função pode conter qualquer comando que já vimos e que vamos ver (inclusive chamadas a outras funções!), exceto outra definição de função.
 - Ele pode conter também a definição de **variáveis locais**, que são variáveis disponíveis apenas dentro da função (além dos parâmetros).

Definindo funções

- O corpo de uma função pode ter *um ou mais* comandos **return**

Definindo funções

- O corpo de uma função pode ter *um ou mais* comandos **return**
 - Uma vez executado, ele termina a execução da função naquele momento, não importando comandos posteriores.

Definindo funções

- O corpo de uma função pode ter *um ou mais* comandos **return**
 - Uma vez executado, ele termina a execução da função naquele momento, não importando comandos posteriores.
 - Se houver alguma expressão na frente de um **return**, esse será o valor que a função devolverá a quem a chamou.

Definindo funções

- O corpo de uma função pode ter *um ou mais* comandos **return**
 - Uma vez executado, ele termina a execução da função naquele momento, não importando comandos posteriores.
 - Se houver alguma expressão na frente de um **return**, esse será o valor que a função devolverá a quem a chamou.
- Se não houver comandos **return** no corpo da função, o último comando indentado é também o último executado.

Definindo funções

- Você pode definir várias funções no seu código, e elas podem chamar umas às outras (com algum cuidado).

Definindo funções

- Você pode definir várias funções no seu código, e elas podem chamar umas às outras (com algum cuidado).
- Um código que não está em uma definição de função é chamado *código global*

Definindo funções

- Você pode definir várias funções no seu código, e elas podem chamar umas às outras (com algum cuidado).
- Um código que não está em uma definição de função é chamado *código global*
 - A definição de uma função deve aparecer antes que qualquer código global a chame.

Definindo funções

- Você pode definir várias funções no seu código, e elas podem chamar umas às outras (com algum cuidado).
- Um código que não está em uma definição de função é chamado *código global*
 - A definição de uma função deve aparecer antes que qualquer código global a chame.
- Em geral, a organização de um código fonte é: definições de todas as funções e então código global (sem intercalá-los)

Exemplo

```
1 def primeira_raiz_bhaskara(a, b, c):
2     discriminante = b*b - 4*a*c
3     d = discriminante ** 0.5
4     return (-b + d) / (2*a)
5
6 a = float(input())
7 b = float(input())
8 c = float(input())
9 print(primeira_raiz_bhaskara(a, b, c))
```

Exemplo

```
1 def primeira_raiz_bhaskara(a, b, c):
2     discriminante = b*b - 4*a*c
3     d = discriminante ** 0.5
4     return (-b + d) / (2*a)
5
6     a = float(input())
7     b = float(input())
8     c = float(input())
9     print(primeira_raiz_bhaskara(a, b, c))
```

Exemplo

```
1 def maior(a, b):
2     z = (x+y)/2 + abs(y-x)/2
3     return z
4
5 def menor(a, b):
6     return (a + b) - maior(a, b)
7
8 x = float(input())
9 y = float(input())
10 print("O menor dentre os números lidos é o " + str(menor(x, y)))
```

Exemplo

```
1 def maior(a, b):
2     z = (a + b)/2 + abs(b - a)/2
3     return z
4
5 def menor(a, b):
6     return (a + b) - maior(a, b)
7
8 x = float(input())
9 y = float(input())
10 menor = menor(x, y)
11 print("O menor dentre os números lidos é o " + str(menor))
```

Escopo de variáveis

Escopo de variáveis

- O *escopo* de uma variável determina de quais partes do código ela pode ser acessada.

Escopo de variáveis

- O *escopo* de uma variável determina de quais partes do código ela pode ser acessada.
- Os parâmetros e variáveis locais de uma função têm escopo limitado àquela função.

Escopo de variáveis

- O *escopo* de uma variável determina de quais partes do código ela pode ser acessada.
- Os parâmetros e variáveis locais de uma função têm escopo limitado àquela função.
- Variáveis definidas no código global (variáveis globais), têm escopo limitado ao arquivo `.py` que as contém.

Escopo de variáveis

- O *escopo* de uma variável determina de quais partes do código ela pode ser acessada.
- Os parâmetros e variáveis locais de uma função têm escopo limitado àquela função.
- Variáveis definidas no código global (variáveis globais), têm escopo limitado ao arquivo `.py` que as contém.
- Assim:

Escopo de variáveis

- O *escopo* de uma variável determina de quais partes do código ela pode ser acessada.
- Os parâmetros e variáveis locais de uma função têm escopo limitado àquela função.
- Variáveis definidas no código global (variáveis globais), têm escopo limitado ao arquivo `.py` que as contém.
- Assim:
 - Código global não consegue usar variáveis locais ou parâmetros de uma função.

Escopo de variáveis

- O **escopo** de uma variável determina de quais partes do código ela pode ser acessada.
- Os parâmetros e variáveis locais de uma função têm escopo limitado àquela função.
- Variáveis definidas no código global (variáveis globais), têm escopo limitado ao arquivo **.py** que as contêm.
- Assim:
 - Código global não consegue usar variáveis locais ou parâmetros de uma função.
 - Uma função não consegue usar variáveis locais ou parâmetros de outra função.

Escopo de variáveis

- O **escopo** de uma variável determina de quais partes do código ela pode ser acessada.
- Os parâmetros e variáveis locais de uma função têm escopo limitado àquela função.
- Variáveis definidas no código global (variáveis globais), têm escopo limitado ao arquivo `.py` que as contém.
- Assim:
 - Código global não consegue usar variáveis locais ou parâmetros de uma função.
 - Uma função não consegue usar variáveis locais ou parâmetros de outra função.
- Em Python, se uma função define um parâmetro ou variável local que tem *o mesmo identificador* de uma variável global, então aquele identificador passa a ser local e não se refere mais à variável global.

Escopo de variáveis

- O **escopo** de uma variável determina de quais partes do código ela pode ser acessada.
- Os parâmetros e variáveis locais de uma função têm escopo limitado àquela função.
- Variáveis definidas no código global (variáveis globais), têm escopo limitado ao arquivo `.py` que as contém.
- Assim:
 - Código global não consegue usar variáveis locais ou parâmetros de uma função.
 - Uma função não consegue usar variáveis locais ou parâmetros de outra função.
- Em Python, se uma função define um parâmetro ou variável local que tem *o mesmo identificador* de uma variável global, então aquele identificador passa a ser local e não se refere mais à variável global.
 - A variável global continua existindo!

Escopo de variáveis

- O **escopo** de uma variável determina de quais partes do código ela pode ser acessada.
- Os parâmetros e variáveis locais de uma função têm escopo limitado àquela função.
- Variáveis definidas no código global (variáveis globais), têm escopo limitado ao arquivo **.py** que as contêm.
- Assim:
 - Código global não consegue usar variáveis locais ou parâmetros de uma função.
 - Uma função não consegue usar variáveis locais ou parâmetros de outra função.
- Em Python, se uma função define um parâmetro ou variável local que tem *o mesmo identificador* de uma variável global, então aquele identificador passa a ser local e não se refere mais à variável global.
 - A variável global continua existindo!
- Para que uma função consiga usar uma variável global, ela deve declarar isso em seu corpo, usando a palavra **global**

Exemplo

```
1 def fun(x):  
2     print(x)  
3     x = 99  
4  
5 x = 4  
6 fun(7)  
7 print(x)
```

Exemplo

```
1 def fun(x):  
2     print(x)  
3     x = 99  
4  
5 x = 4  
6 fun(7)  
7 print(x)
```

```
1 def fun(y):  
2     global x  
3     x = 99  
4  
5 x = 4  
6 fun(7)  
7 print(x)
```

Variáveis globais

- Apesar de ser possível que uma função acesse uma variável global, evite fazer isso!

Variáveis globais

- Apesar de ser possível que uma função acesse uma variável global, evite fazer isso!
 - A legibilidade do seu código pode piorar com o uso de variáveis globais

Variáveis globais

- Apesar de ser possível que uma função acesse uma variável global, evite fazer isso!
 - A legibilidade do seu código pode piorar com o uso de variáveis globais
 - Usá-las acaba causando uma dependência entre partes distintas de código, e funções deveriam nos ajudar justamente a criar partes independentes.

Variáveis globais

- Apesar de ser possível que uma função acesse uma variável global, evite fazer isso!
 - A legibilidade do seu código pode piorar com o uso de variáveis globais
 - Usá-las acaba causando uma dependência entre partes distintas de código, e funções deveriam nos ajudar justamente a criar partes independentes.
- Basta fazer uso de parâmetros e argumentos corretamente.

Fluxo de execução

Fluxo de execução

- O *fluxo de execução* de um programa tem início em algum ponto do código e segue sequencialmente os comandos a partir dali.

Fluxo de execução

- O *fluxo de execução* de um programa tem início em algum ponto do código e segue sequencialmente os comandos a partir dali.
- Quando uma função é chamada, o fluxo de execução muda, sendo desviado para a primeira linha do corpo da função.

Fluxo de execução

- O *fluxo de execução* de um programa tem início em algum ponto do código e segue sequencialmente os comandos a partir dali.
- Quando uma função é chamada, o fluxo de execução muda, sendo desviado para a primeira linha do corpo da função.
 - Copia-se o valor de cada argumento, na ordem, para cada parâmetro.

Fluxo de execução

- O *fluxo de execução* de um programa tem início em algum ponto do código e segue sequencialmente os comandos a partir dali.
- Quando uma função é chamada, o fluxo de execução muda, sendo desviado para a primeira linha do corpo da função.
 - Copia-se o valor de cada argumento, na ordem, para cada parâmetro.
 - Executa-se cada linha do corpo da função, uma após a outra.

Fluxo de execução

- O *fluxo de execução* de um programa tem início em algum ponto do código e segue sequencialmente os comandos a partir dali.
- Quando uma função é chamada, o fluxo de execução muda, sendo desviado para a primeira linha do corpo da função.
 - Copia-se o valor de cada argumento, na ordem, para cada parâmetro.
 - Executa-se cada linha do corpo da função, uma após a outra.
 - Retorna-se o fluxo para o ponto exato onde a função foi chamada.

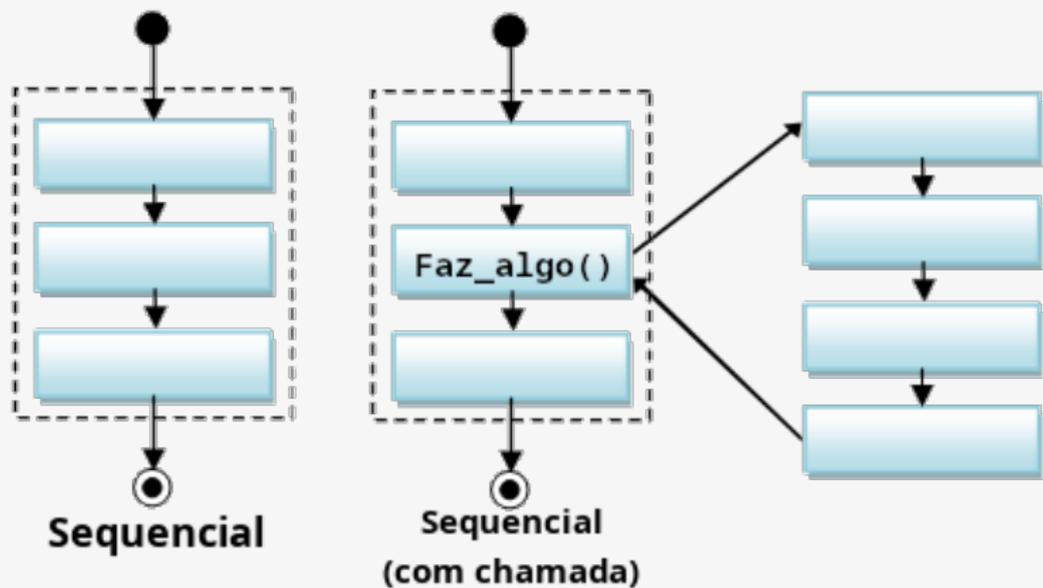
Fluxo de execução

- O *fluxo de execução* de um programa tem início em algum ponto do código e segue sequencialmente os comandos a partir dali.
- Quando uma função é chamada, o fluxo de execução muda, sendo desviado para a primeira linha do corpo da função.
 - Copia-se o valor de cada argumento, na ordem, para cada parâmetro.
 - Executa-se cada linha do corpo da função, uma após a outra.
 - Retorna-se o fluxo para o ponto exato onde a função foi chamada.
 - O valor devolvido é substituído na expressão em que a chamada apareceu.

Fluxo de execução

- O *fluxo de execução* de um programa tem início em algum ponto do código e segue sequencialmente os comandos a partir dali.
- Quando uma função é chamada, o fluxo de execução muda, sendo desviado para a primeira linha do corpo da função.
 - Copia-se o valor de cada argumento, na ordem, para cada parâmetro.
 - Executa-se cada linha do corpo da função, uma após a outra.
 - Retorna-se o fluxo para o ponto exato onde a função foi chamada.
 - O valor devolvido é substituído na expressão em que a chamada apareceu.
- É importante observar também que uma função pode chamar outra!

Fluxo de execução



Exemplo

```
1 def proximo_impar(num):
2     """Devolve o menor inteiro ímpar que é maior ou igual a num.
3     num deve ser um inteiro
4     """
5     return num + (1 - num%2)
6
7 x = int(input())
8 print(proximo_impar(x))
```

Resumo

Resumo

A partir de agora, sempre que você perceber que é possível separar a resolução de um problema em tarefas menores, separe em funções!

Exemplo

Um exemplo com tudo

Faça um programa que recebe as coordenadas x e y de três pontos e encontra a distância dos dois mais próximos.

Pratique!

Exercícios

1. Qual a saída do seguinte código?

```
1  def b(z):
2      prod = a(z, z)
3      print(z, prod)
4      return prod
5
6  def a(x, y):
7      x = x + 1
8      return x * y
9
10 def c(x, y, z):
11     total = x + y + z
12     quadr = b(total)**2
13     return quadr
14
15 x = 1
16 y = x + 1
17 print(c(x, y+3, x+y))
```

Exercícios

IMPORTANTE: Não use o operador de seleção `if` em sua solução e nem laços! Use funções!

1. Escreva um programa que recebe três inteiros positivos e escreve **False** se algum deles é maior ou igual à soma dos outros dois e **True** caso contrário. (Isso faz parte da verificação se três dados números podem ser os comprimentos dos lados de um triângulo.)
2. Escreva um programa que recebe dois inteiros i e f , com $i \leq f$, e que calcula o valor final da expressão

$$\sum_{k=i}^f k.$$

Dica: sabemos que $\sum_{k=0}^n k = \frac{n(n+1)}{2}$.