Algoritmos e Programação de Computadores

Revisão: Prova 1

Profa. Sandra Avila
Instituto de Computação (IC/Unicamp)

MC102, 30 Abril, 2020

Conteúdo da Prova 1

- Variáveis & Tipos (`int`, `float`, `str`, `bool`)
- Operações & Expressões aritméticas, relacionais, lógicas
- Comandos condicionais (`if-elif-else`)
- Comandos repetitivos (`for`, `while`)
- Listas
- Strings

Revisão do Conteúdo

Objetos

- Qualquer dado em Python é um **objeto**, que é de um certo **tipo** específico.
- O **tipo** de um objeto especifica quais operações podem ser realizadas sobre o objeto.

Objetos

```
print(type("Olá turma de MC102"))
print(type(5))
```

```
<class 'str'>
<class 'int'>
```

"Olá turma de MC102" é uma **string ou texto cadeia de caracteres**, do tipo **str**

5 é um **inteiro**, do tipo **int**

Objetos

```
print(type("5"))
```

```
<class 'str'>
```

5 é um número inteiro, mas como está entre aspas é uma string.

Variáveis

- Variáveis são uma forma de se associar um nome dado pelo programador com um objeto.

```
altura = 10
nota = 9.75
turma = "GHI"
```

Variáveis: Regras para Nomes

- **Deve** começar com uma letra (maiúscula ou minúscula) ou underscore(_). **Nunca** pode começar com um número.
- Pode conter letras maiúsculas, minúsculas, números e subscrito.
- Não pode-se utilizar como parte do nome de uma variável:

  { (+ - * / \ n ; . , ? $

- Letras maiúsculas e minúsculas são diferentes: c = 4 C = 3

Variáveis: Regras para Nomes

```
102MC = "disciplina legal"
mais$ = 1000000
class = "MC102"
```

O nome `102MC` é ilegal pois **não começa com uma letra**.

`mais$` é ilegal pois contém um **caractere ilegal**, o símbolo de cifrão.

`class` é uma palavra reservada em Python.

Variáveis: Palavras Reservadas

and	as	assert	break	class	continue
def	del	elif	else	except	exec
finally	for	from	global	if	import
in	is	lambda	nonlocal	not	or
ass	raise	return	try	while	with
yield	True	False	None		

Variáveis: Palavras Reservadas

and	as	assert	break	class	continue
def	del	elif	else	except	exec
finally	for	from	global	if	import
in	is	lambda	nonlocal	not	or
ass	raise	return	try	while	with
yield	True	False	None		

O que vimos até agora :)

Atribuição

- O comando = do Python é o comando de atribuição. Ele associa a variável do lado esquerdo do comando com o objeto do lado direito do comando.

Atribuição

- O comando = do Python é o comando de atribuição. Ele associa a variável do lado esquerdo do comando com o objeto do lado direito do comando.

Lembrete! = é atribuição == é igualdade

Atribuição

- Se uma variável for usada sem estar associada com nenhum objeto, um erro ocorre.
- No exemplo abaixo **não** podemos usar a variável `c`, pois esta **não** foi definida (associada com algum objeto).

```
a = 10
b = 10
a = a + b
a
20
a = a + c
```

Tipos

- **int**: números inteiros.
  Ex: 10, -24.
- **float**: números decimais.
  Ex: 2.4142, 3.141592.
- **str** ou **string**: textos.
  Ex: "Olá turma".
- **bool**: valores booleanos falso (`False`) e verdadeiro (`True`).

Tipo Inteiro

- Objetos do tipo **int** armazenam valores inteiros.
- Literais do tipo **int** são escritos comumente como escrevemos inteiros.
- Exemplos: 3, 1034, e -512.
- O tipo **int** possui precisão arbitrária (limitado a memória do seu computador).

Tipo Ponto Flutuante

- Objetos do tipo **float** armazenam valores “reais”.
- Literais do tipo **float** são escritos com um ponto para separar a parte inteira da parte decimal. Exemplos: 3.1415 e 9.8.
- Possuem problemas de precisão pois há uma quantidade limitada de memória para armazenar um número real no computador.

Tipo String

- Objetos do tipo **string** armazenam textos.
- Um literal do tipo **string** deve estar entre aspas simples ou aspas duplas. Exemplos de **strings**:
  - 'Olá Pessoal!' ou "Olá Pessoal".

Tipo Bool

- Em Python o tipo **bool** especifica os valores booleanos falso (`False`) e verdadeiro (`True`).
- Podemos criar variáveis associadas a booleanos, mas o uso mais comum é na verificação de resultados de expressões relacionais e lógicas.

```
a = True
print(type(a))
<class 'bool'>
```

Tipagem em Python

- Uma variável em Python possui o tipo correspondente ao objeto que ela está associada **naquele instante**.
- Python não possui tipagem forte como outras linguagens.
  - Isto significa que você pode atribuir objetos de diferentes tipos para uma mesma variável.
  - Como uma variável não possui tipo pré-definido, dizemos que Python tem **tipagem fraca**.
  - Em outras linguagens cria-se variáveis de tipos específicos e elas só podem armazenar valores daquele tipo para o qual foram criadas.
  - Estas últimas linguagens possuem **tipagem forte**.

Tipagem em Python

```
a = 3
print(a)
3
a = 90.45
print(a)
90.45
a = "Olá vocês!"
print(a)
Olá vocês!
a = True
print(a)
True
```

Operações Básicas

- Saída de dados: `print()`
- Entrada de dados: `input()`
- Expressões
- Operadores Aritméticos

A Função print()

Escrevendo na Tela: `print()`

- Para imprimir um texto, utilizamos o comando **`print()`**.
- O texto pode ser um literal do tipo **string**.

```
print("De novo isso?")
De novo isso?
```

- No meio da **string** pode-se incluir caracteres de formatação especiais.
- O símbolo especial `\n` é responsável por pular uma linha na saída.

```
print("De novo isso? \n Olá Pessoal!")
De novo isso?
Olá Pessoal!
```

Escrevendo o Conteúdo de uma Variável na Tela

- Podemos imprimir, além de texto puro, o conteúdo de uma variável utilizando o comando **`print()`**.
- Separamos múltiplos argumentos a serem impressos com uma vírgula.

```
a = 10
print("A variável contém o valor", a)
A variável contém o valor 10
```

Escrevendo o Conteúdo de uma Variável na Tela

- A impressão com múltiplos argumentos inclui um **espaço extra** entre cada argumento.

```
a = 10
b = 3.14
print("a contém o valor", a, "e b contém o valor", b)
a contém o valor 10 e b contém o valor 3.14
```

Escrevendo o Conteúdo de uma Variável na Tela

- Podemos converter todos os valores em strings e usar o **operador +** para concatenar strings de forma a imprimir sem estes espaços:

```
a = 10
b = 3.14
print("a contém o valor " + str(a) + " e b contém o valor " + str(b))
a contém o valor 10 e b contém o valor 3.14
```

A Função input()

A Função `input()`

- Realiza a leitura de dados a partir do teclado.
- Aguarda que o usuário digite um valor e atribui o valor digitado a uma variável.
- **Todos** os dados lidos são do tipo string.

```
print("Digite um número:")
numero = input()
print("O número digitado é:" + numero)
```

A Função `input()`

- Podemos converter uma string lida do teclado em um número inteiro usando a função **int()** (ou um número float usando a função **float()**).

```
print("Digite um número:")
numero = int(input())
numero = numero * 10
print("O número digitado vezes 10 é:", numero)
```

Expressões Aritméticas

Expressões Aritméticas

- Os operadores aritméticos são: `+`, `-`, `*`, `/`, `//`, `%`, `**`

- **Adição**: expressão `+` expressão
- **Subtração**: expressão `-` expressão
- **Multiplicação**: expressão `*` expressão
- **Divisão**: expressão `/` expressão: resultado é sempre um float.
- **Divisão**: expressão `//` expressão: se os operandos forem inteiros, a divisão é inteira. Se um deles for ponto flutuante faz uma divisão truncada.
- **Resto da Divisão**: expressão `%` expressão
- **Exponenciação (potenciação)**: expressão `**` expressão

Precedência

- Precedência é a **ordem** na qual os operadores serão avaliados quando o programa for executado.
- Em Python, os operadores são avaliados na seguinte ordem:
  1. **
  2. *, /, //, na ordem em aparecerem na expressão
  3. %
  4. +, -, na ordem em aparecerem na expressão
- Exemplo: 8 + 10 * 6 é igual a 68

Alterando a Precedência

- **(expressão)** também é uma expressão, que calcula o resultado da expressão dentro dos parênteses, para só então calcular o resultado das outras expressões.
  - 5 + 10 % 3 é igual a 6
  - (5 + 10) % 3 é igual a 0
- Você pode usar quantos parênteses desejar dentro de uma expressão.
- Use sempre parênteses em expressões para deixar claro em qual ordem a expressão é avaliada!

Expressões Relacionais

Expressões Relacionais

- Expressões relacionais são aquelas que realizam uma **comparação** entre duas expressões e retornam
  - `False`, se o resultado é falso.
  - `True`, se o resultado é verdadeiro.

Operadores Relacionais

- Os operadores relacionais da linguagem Python são:
  - `==` : igualdade
  - `!=` : diferente
  - `>` : maior que
  - `<` : menor que
  - `>=` : maior ou igual que
  - `<=` : menor ou igual que

- expressão `==` expressão : Retorna verdadeiro quando as expressões forem iguais.
- expressão `!=` expressão : Retorna verdadeiro quando as expressões forem diferentes.
- expressão `>` expressão : Retorna verdadeiro quando a expressão da esquerda tiver valor maior que a expressão da direita.
- expressão `<` expressão : Retorna verdadeiro quando a expressão da esquerda tiver valor menor que a expressão da direita.
- expressão `>=` expressão : Retorna verdadeiro quando a expressão da esquerda tiver valor maior ou igual que a expressão da direita.
- expressão `<=` expressão : Retorna verdadeiro quando a expressão da esquerda tiver valor menor ou igual que a expressão da direita.

Expressões Lógicas

Expressões Lógicas

- Expressões lógicas são aquelas que realizam uma operação lógica (**ou**, **e**, **não**, etc...) e retornam `True` ou `False` (como as expressões relacionais).

- Na linguagem Python temos os seguintes operadores lógicos:
  - **`and`** : operador E
  - **`or`**: operador OU
  - **`not`**: operador NÃO

Expressões Lógicas

- expressão `and` expressão : Retorna verdadeiro quando **ambas** as expressões são verdadeiras.
- expressão `or` expressão : Retorna verdadeiro quando **pelo menos uma** das expressões é verdadeira.
- `not` expressão : Retorna verdadeiro quando a expressão é falsa e vice-versa.

Expressões Lógicas

```
print(8 > 9 and 10 != 2)
print(14 > 100 or 2 > 1)
print(not(14 > 100) and not(1 > 2))
```

```
False
True
True
```

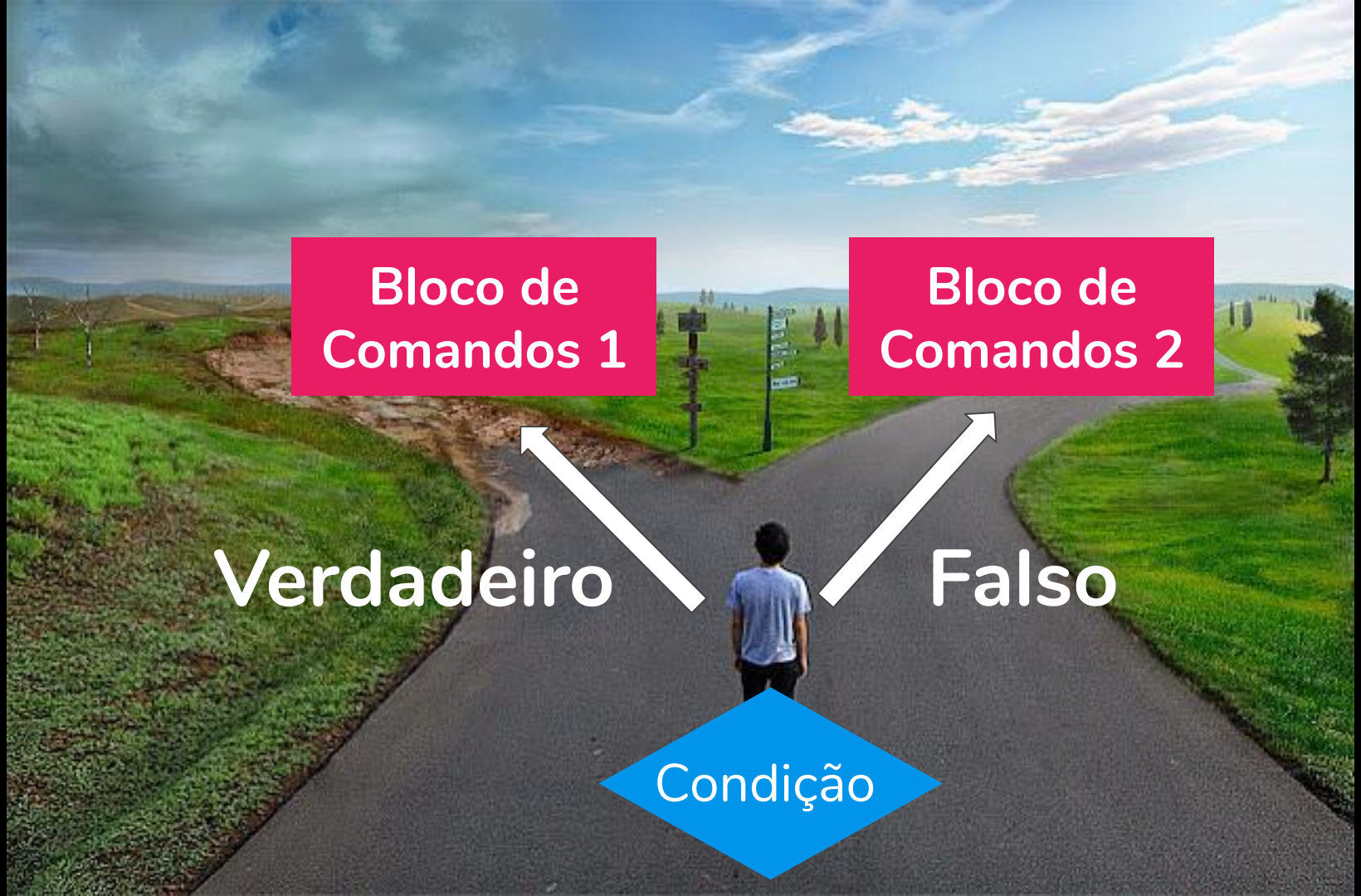
Precedência de Operadores

Nível	Categoria	Operadores
7 (alto)	exponenciação	`**`
6	multiplicação	`*`, `/`, `//`, `%`
5	adição	`+`, `-`
4	relacional	`==`, `!=`, `<=`, `>=`, `>`, `<`
3	lógico	`not`
2	lógico	`and`
1 (baixo)	lógico	`or`

Comandos Condicionais

- Permite decidir **se** um determinado bloco de comandos deve ou não ser executado, a partir do resultado de uma expressão relacional ou lógica.

- `if`, `elif`, `else`


Bloco de Comandos 1
Bloco de Comandos 2
Verdadeiro
Falso
Condição

Comandos Condicionais

- O principal comando condicional é o **if**, cuja sintaxe é:

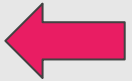
```
if expressão relacional ou lógica:
    comandos executados se a expressão é verdadeira
```

- Os comandos são executados somente se a expressão relacional/lógica for verdadeira.

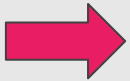
Comandos Condicionais

- O principal comando condicional é o **if**, cuja sintaxe é:

```
if expressão relacional ou lógica:
    comandos executados se a expressão é verdadeira
```



dois pontos



- Os comandos são executados somente se a expressão relacional/lógica for verdadeira.

Comandos Condicionais

- Uma variação do comando **if** é o **if/else**, cuja sintaxe é:

```
if expressão relacional ou lógica:
    comandos executados se a expressão é verdadeira
else:
    comandos executados se a expressão é falsa
```

dois pontos

dois pontos

comandos indentados

Comandos Condicionais

```
if (condicao1):
    comandos1
```

```
if (condicao1):
    comandos1
else:
    comandos2
```

Comandos Condicionais

```
if (condicao1):
    if (condicao2):
        comandos1
    else:
        comandos2
else:
    if (condicao3):
        comandos3
    else:
        comandos4
```

Comandos Condicionais

- O programa determina o menor de dois números.

```
# Determina o menor de dois números.
numero1 = int(input("Digite um número:"))
numero2 = int(input("Digite um número:"))

if numero1 < numero2:
    print("O menor número é:", numero1)
else:
    print("O menor número é:", numero2)
```

Comandos Condicionais

```
numero = 5
if (numero > 3):
    if (numero < 7):
        print("a")
else:
    if (numero > -10):
        print("b")
    else:
        print("c")
```

Comandos Condicionais

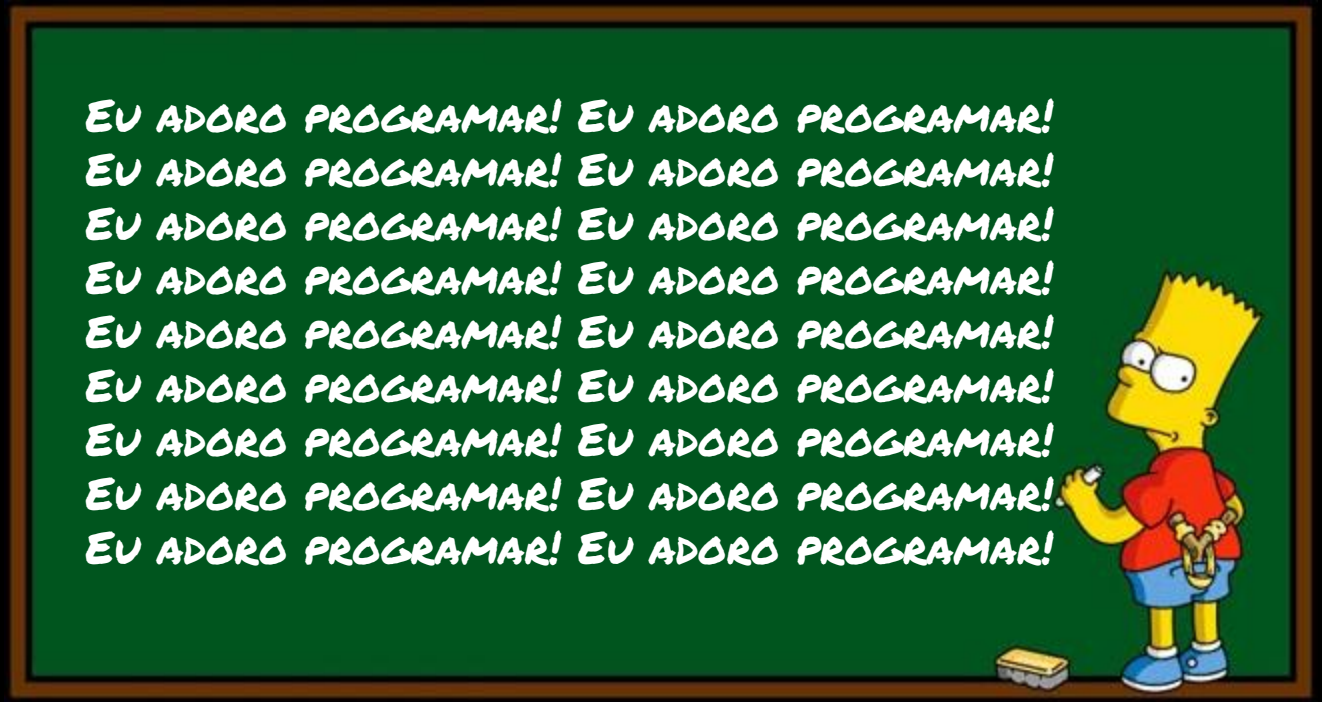
Quando **apenas uma** de várias alternativas é verdadeira podemos usar a construção `if-elif-else` que em Python é representado por:

```
if condicao1:
  comandos1
elif condicao2:
  comandos2
elif condicao3:
  comandos3
elif condicao4:
  comandos4
else:
  comandos5
```

n vezes

Comandos Repetitivos

- Permite executar um bloco de comandos várias vezes para obter o resultado esperado.
- Comandos repetitivos, iterativos, laços, loops, ...
- `while`, `for`


EU ADORO PROGRAMAR! EU ADORO PROGRAMAR!
EU ADORO PROGRAMAR! EU ADORO PROGRAMAR!
EU ADORO PROGRAMAR! EU ADORO PROGRAMAR!
EU ADORO PROGRAMAR! EU ADORO PROGRAMAR!
EU ADORO PROGRAMAR! EU ADORO PROGRAMAR!
EU ADORO PROGRAMAR! EU ADORO PROGRAMAR!
EU ADORO PROGRAMAR! EU ADORO PROGRAMAR!
EU ADORO PROGRAMAR! EU ADORO PROGRAMAR!
EU ADORO PROGRAMAR! EU ADORO PROGRAMAR!

Comando `while`

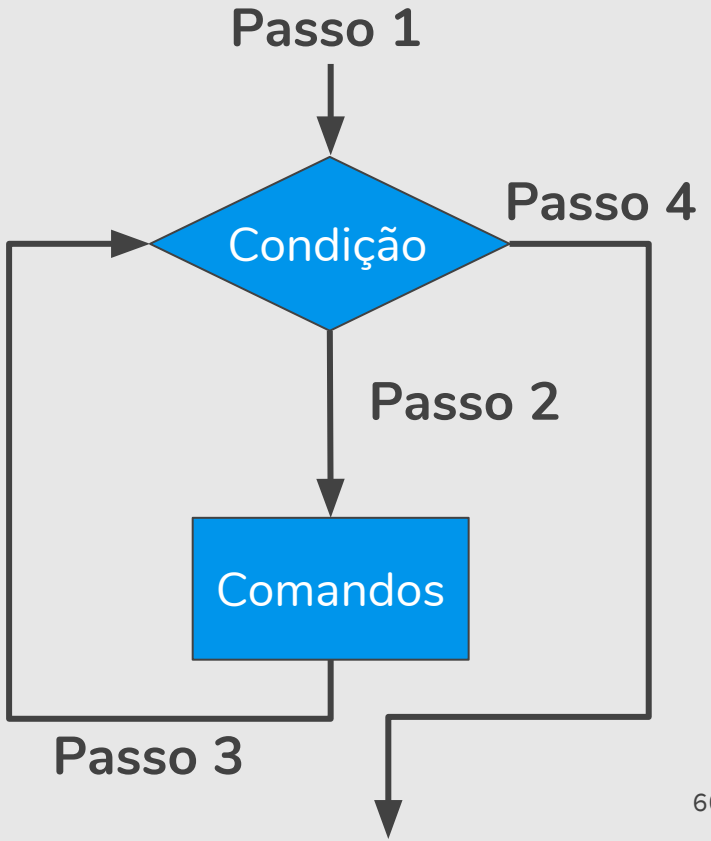
Comando `while`

- Executa um bloco de comando(s) enquanto a condição é verdadeira (`True`).

```
while condicao:
    comandos
```

Comando `while`

- **Passo 1**: Testa condição.
  - Se condição for verdadeira, vai para o **Passo 2**
  - Senão, vai para **Passo 4**
- **Passo 2**: Executa comandos
- **Passo 3**: Volta para **Passo 1**



Comando `while`

- Programa que imprime os *n* primeiros números.

```
# Imprime os n primeiros números
n = int(input("Digite um número: "))
numero = 1
while numero <= n:
    print(numero)
    numero = numero + 1
```

Comando `for`

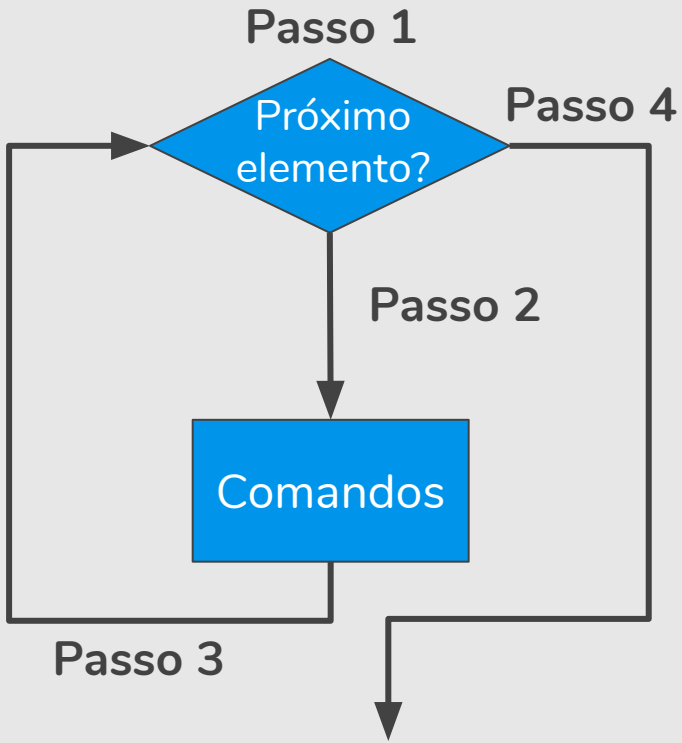
Comando `for`

- É a estrutura de repetição mais usada no Python.
- Para cada elemento da lista, em ordem de ocorrência, é atribuído este elemento à variável e então é executado o(s) comando(s).

```
for variável in lista:
    comando(s)
```

Comando `for`

- **Passo 1**: Verifica se percorreu toda a lista.
  - Se não percorreu, atribui-se o próximo elemento da lista para a variável.
  - Se percorreu, vai para **Passo 4**
- **Passo 2**: Executa comandos
- **Passo 3**: Volta para **Passo 1**



A Função `range`

- É comum fazermos um laço `for` iterar sobre valores numéricos.
- Em Python, a função `range(n)` gera uma lista com valores de `0` até $n-1$.
- Programa que imprime todos os números de 0 a 9.

```
# Imprime todos os números de 0 a 9
for numero in range(10):
    print(numero)
```

A Função `range`

- Podemos especificar um passo a ser considerado no intervalo de valores na função `range(n)`
  - `range(inicio,fim,passo)`: gera-se números de `inicio` com incremento de `passo` até *`fim`*`-1`.

A Função `range`

- Programa que imprime todos os números pares entre 0 e 13.

```
# Imprime todos os números pares entre 0 e 13
for numero in range(0,13,2):
    print(numero)
```

```
0
2
4
6
8
10
12
```

while e for

- Programa que imprime os *n* primeiros números.

```
# Imprime os n primeiros números
n = int(input("Digite um número: "))
numero = 1
while numero <= n:
    print(numero)
    numero = numero + 1
```

```
# Imprime os n primeiros números
n = int(input("Digite um número: "))
for numero in range(1,n+1):
    print(numero)
```

`while` ou `for`?

- `for` : “iteração definida”
- `while` : “iteração indefinida”, não temos certeza de quantas iterações precisamos nem podemos estabelecer um limite superior.

Laços e o Comando `break`

- O comando `break` faz com que a execução de um laço seja terminada, passando a execução para o próximo comando depois do final do laço.

```
while condicao:
    comando(s)
    break
comando(s)
```

```
for variável in lista:
    comando(s)
    break
comando(s)
```

Laços Encaixados

- Para resolver alguns problemas, é necessário implementar um laço dentro de outro laço.
- Estes são laços encaixados.

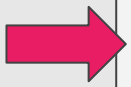
```
for i in range(1,11):
    for j in range(1,6):
        print(i, j)
```

Variável Indicadora

Variável Indicadora

- Um uso comum de laços é para a verificação se um determinado objeto, ou conjunto de objetos, satisfaz uma propriedade ou não.
- Um padrão que pode ser útil na resolução deste tipo de problema é o uso de uma **variável indicadora**.
  - Assumimos que o objeto satisfaz a propriedade (**indicadora = `True`**).
  - Com um laço verificamos se o objeto realmente satisfaz a propriedade.
  - Se em alguma iteração descobrirmos que o objeto não satisfaz a propriedade, então fazemos **indicadora = `False`**.

Exemplo: Número Primo



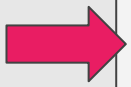
```
n = int(input("Digite um número inteiro positivo: "))

numero = 2
primo = True # primo é a variável indicadora

while (numero <= n-1) and (primo):
    if (n % numero == 0): # se n é divisível por numero
        primo = False
    numero = numero + 1

if (primo):
    print("É primo.")
else:
    print("Não é primo.")
```

Variável Contadora

Variável Contadora

- Considere ainda o uso de laços para a verificação se um determinado objeto, ou conjunto de objetos, satisfaz uma propriedade ou não.
- Um outro padrão que pode ser útil é o uso de uma **variável contadora**.
    - Esperamos que um objeto satisfaça x vezes uma sub-propriedade. Usamos um laço e uma variável que **conta** o número de vezes que o objeto tem a sub-propriedade satisfeita.
    - Ao terminar o laço, se a variável contadora for igual à x então o objeto satisfaz a propriedade.

Exemplo: Número Primo



```
n = int(input("Digite um número inteiro positivo: "))

numero = 2
divisores = 0 # divisores é a variável contadora

while (numero <= n-1) and (divisores == 0):
    if (n % numero == 0): # se n é divisível por numero
        divisores = divisores + 1
    numero = numero + 1

if (divisores == 0):
    print("É primo.")
else:
    print("Não é primo.")
```

Variável Acumuladora

Variável Acumuladora

- Vamos ver alguns exemplos de problemas que são resolvidos utilizando laços.
- Há alguns padrões de solução que são bem conhecidos, e são úteis em diversas situações.
- O primeiro padrão deles é o uso de uma “variável acumuladora”.

Soma de Números

- Como *n* não é definido a priori, não podemos criar *n* variáveis e depois somá-las.
- A ideia é criar uma variável acumuladora que a cada iteração de um laço **acumula** a soma de todos os números lidos até então.

```
acumuladora = 0
repita n vezes
    leia um número aux
    acumuladora = acumuladora + aux
```

Soma de Números

- Programa que soma *n* números.

```
# Soma n números
n = int(input("Digite o valor de n: "))
acumuladora = 0
for numero in range(n):
    aux = int(input())
    acumuladora = acumuladora + aux   # Acumula a soma
print("A soma é:", acumuladora)
```

Listas

- Servem para armazenar vários dados de forma simplificada.
- `lista = [2,"mc102", 9.75, [2,4], "b"]`

Exemplos de Listas

- Lista de inteiros:

```
x = [2, 45, 12, 9, -2]
```

- Listas podem conter dados de tipos diferentes:

```
x = [2,"qwerty" ,45.99087, 0, "a"]
```

- Listas podem conter outras listas:

```
x = [2, [4,5], [9]]
```

- Ou podem não conter nada. Neste caso `[]` indica a lista vazia.

Listas: Como Usar

- Pode-se acessar uma determinada posição da lista utilizando-se um índice de valor inteiro.
- A sintaxe para acesso de uma determinada posição é:
  - `identificador[`**`posição`**`]`
- Sendo *n* o tamanho da lista, os índices válidos para ela vão de 0 até *n* – 1.
  - A primeira posição da lista tem índice 0.
  - A última posição da lista tem índice *n* – 1.

Listas: Índices

- O **slicing** em Python é obtido como

  `identificador[ind1:ind2]`

  e o resultado é uma sub-lista com os elementos de `ind1` até `ind2`−1.

```
notas = [8.0, 5.5, 9.3, 0.5, 3.1]
print(notas[1:4])
```

```
[5.5, 9.3, 0.5]
```

Listas: Função `len`

- A função **`len`**(lista) retorna o número de itens na lista.

```
notas = [8.0, 5.5, 9.3, 0.5, 3.1]
len(notas)
```

```
5
```

- É muito comum usar a função **`len`** junto com o laço **`for`** para percorrer todas as posições de uma lista:

```
notas = [8.0, 5.5, 9.3, 0.5, 3.1]
for i in range(len(notas)):
    print(notas[i])
```

Listas: `for`

- Lembre-se que o **`for`** na verdade faz a variável de controle assumir todos os valores de uma lista. Assim:

```
notas = [8.0, 5.5, 9.3, 0.5, 3.1]
for i in range(len(notas)):
    print(notas[i])
```

- E também pode ser implementado como:

```
notas = [8.0, 5.5, 9.3, 0.5, 3.1]
for i in notas:
    print(i)
```

Listas: append

- Uma operação importante é acrescentar um item no final de uma lista. Isto é feito pela função **append**.

```
lista.append(item)
```

```
notas = [8.0, 5.5, 9.3, 0.5, 3.1]
notas.append(9.5)
print(notas)
```

```
[8.0, 5.5, 9.3, 0.5, 3.1, 9.5]
```

Strings

- Strings em Python são listas imutáveis de caracteres.
- Strings são representadas por sequências de caracteres entre aspas simples ' ou entre aspas duplas ".

Strings: Resumo

Método	Parâmetros	Descrição
`strip`	nenhum	Retorna uma string removendo caracteres em branco do início e do fim. Ex: `a.strip()`
`find`	substring	Retorna o índice onde a substring começa na string. Ex: `a.find("texto")`
`split`	nenhum	Separa uma string usando sep como separador e retorna uma lista das substrings. Ex: `a.split()`
`replace`	substring1, substring2	Substitui todas as ocorrências de uma substring por outra. Ex: `a.replace("prova","teste")`
`list`	substring	Transforma uma string em uma lista onde os itens da lista correspondem aos caracteres da string. Ex: `list("texto")` ou `list(a)`
`join`	substring	Retorna uma string com a concatenação dos elementos da sequência/lista. Ex: `"".join(a)`
`count`	substring	Retorna o número de ocorrências de uma substring. Ex: `a.count("as")`
`upper`	nenhum	Retorna uma string toda em maiúsculas. Ex: `a.upper()`
`lower`	nenhum	Retorna uma string toda em minúsculas. Ex: `a.lower()`

Exercícios

1. Qual valor é exibido pelo seguinte comando:

```
print(int(53.785))
```

a. Nada, é produzido um erro de execução.
b. 53
c. `54`
d. `53.785`

1. Qual valor é exibido pelo seguinte comando:

```
print(int(53.785))
```

a. Nada, é produzido um erro de execução.

b. `53`

c. `54`

d. `53.785`

2. Qual é o valor impresso ao final da seguinte sequência de comandos?

```
dia = "sexta-feira"
dia = 32.5
dia = 19
print(dia)
```

a. Nada é impresso, ocorre um erro de execução.

b. `sexta-feira`

c. `32.5`

d. `19`

2. Qual é o valor impresso ao final da seguinte sequência de comandos?

```
dia = "sexta-feira"
dia = 32.5
dia = 19
print(dia)
```

a. Nada é impresso, ocorre um erro de execução.

b. `sexta-feira`

c. `32.5`

d. `19`

3. O que imprime o seguinte comando?

```
print(18/4, 18//4, 18%4)
```

a. `4 4.5 2`

b. `4 4 4.5`

c. `4.5 4 2`

d. `4.4 4.25 2`

3. O que imprime o seguinte comando?

```
print(18/4, 18//4, 18%4)
```

a. `4 4.5 2`

b. `4 4 4.5`

c. `4.5 4 2`

d. `4.4 4.25 2`

4. O que imprime o seguinte comando?

```
idade = input("Por favor, digite a sua idade: ")
# usuário digita 18
print(type(idade))
```

a. <class 'str'>

b. <class 'int'>

c. <class 18>

d. 18

4. O que imprime o seguinte comando?

```
idade = input("Por favor, digite a sua idade: ")
# usuário digita 18
print(type(idade))
```

a. **<class 'str'>**

b. <class 'int'>

c. <class 18>

d. 18

5. Qual é o valor a expressão a seguir?

```
16 - 2 * 5 // 3 + 1
```

a. 3

b. 24

c. 14

d. 13.667

5. Qual é o valor a expressão a seguir?

```
16 - 2 * 5 // 3 + 1
```

a. `3`

b. `24`

c. `14`

d. `13.667`

Continuação na próxima aula ...