# Algoritmos e estrutura de dados

## Introdução

Marco A L Barbosa

Este trabalho está licenciado com uma Licença Creative Commons - Atribuição-CompartilhaIgual 4.0 Internacional.

# Conteúdo

A linguagem Python

Usando o editor IDLE

Tipos e operações pré-definidas

Bibliotecas

Testes automatizados

Sentenças de seleção

Variáveis

Sentenças de repetição

Listas (arranjos)

Efeitos colaterais e referências

Classes (estruturas)

# A linguagem Python

# A linguagem Python

- Python é uma linguagem de programação de propósito geral simples e ao mesmo tempo poderosa
- Usado por grandes empresas como Google e Yahoo!
- Também é usado por engenheiros para cálculos numéricos (com a biblioteca NumPy entre outras)

# A linguagem Python

- Vantagens em relação ao Pascal, C e C++
  - Sintaxe mais simples
  - Interpretada (é mais fácil interagir e testar os programas)
  - Tipagem dinâmica
  - Gerência automática de memória
  - Biblioteca padrão extensa
- Desvantagens em relação ao Pascal, C e C++
  - Os programas são geralmente mais lentos e consomem mais memória
  - Os erros de tipo são detectados apenas durante a execução do programa

# Instalação

- Página oficial do Python, seção downloads
- Instalar a versão 3.3.2
- Após a instalação, execute o programa IDLE

# Usando o editor IDLE

# Usando o editor IDLE

- Janela de **interações**
  - Digite expressões (pequenos trechos de código), pressione enter, o Python irá avaliar a expressão e exibir o resultado

```
>>> 3 + 4 * 2
11
```

- Janela de **definições**
  - Para fazer novas definições crie um novo arquivo (File -> New File ou atalho `ctrl + N`)
  - Digite as definições e salve o arquivo (File -> Save ou atalho `ctrl + S`)
  - Carregue as novas definições (Run -> Run Module ou atalho `F5`)
  - Teste as novas definições na janela de interações

# Usando o editor IDLE

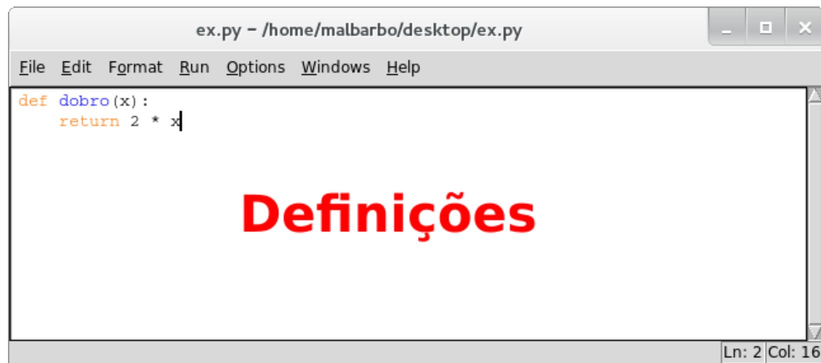


Figura : Janela de definições

# Usando o editor IDLE

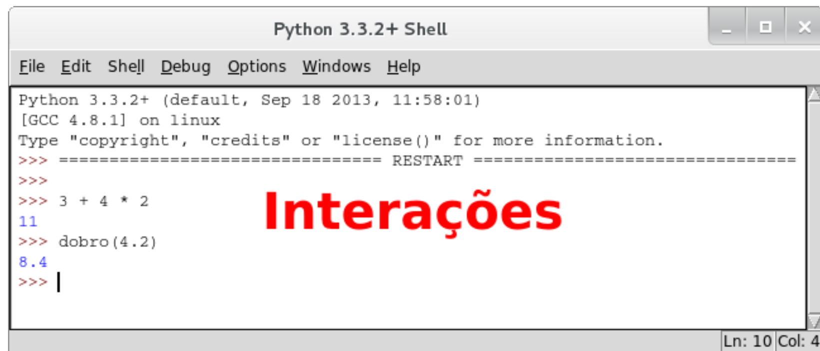


Figura : Janela de interações

# Tipos e operações pré-definidas

# Tipos e operações pré-definidas

- Números
  - Inteiros (`int`)
  - Ponto flutuante (`float`) - representação aproximada de números reais
  - Complexos, frações e decimais (não estudaremos estes tipos)
- Booleano
  - Valor `True` (verdadeiro)
  - Valor `False` (falso)
- String (`str`)
  - Usando para representar nomes e textos em geral
- Sequências, dicionários, etc.

# Tipos e operações pré-definidas

- Operações aritméticas

```
>>> 3 + 2      # soma
5
>>> 4 - 8      # subtração
-4
>>> 3 * 6      # multiplicação
18
>>> 7 / 3      # divisão
2.3333333333333335
>>> 7 // 3     # divisão inteira
2
>>> 7 % 3      # resto da divisão
1
>>> pow(2, 3) # exponenciação
8
>>> 2 ** 3     # exponenciação
8
>>> - 4        # negação
-4
>>> abs(-5)    # valor absoluto
5
```

# Tipos e operações pré-definidas

- O Python utiliza a mesma precedência que estamos acostumados na matemática
- Podemos usar parênteses para mudar a precedência

```
>>> 3 + 4 * 2
11
>>> (3 + 4) * 2
14
>>> 2 + 4 / 2 ** 3
2.5
```

- Qual é o resultado das seguintes expressões?

```
15 // 7
15 % 7
12 // 27
12 % 27
3 * 4 - 5 / 8 // 3
8 / 4 / 2
2 - 4 ** 3 / 9 % 5
```

# Tipos e operações pré-definidas

- Conversões de números

```
>>> int(3.4)   # Transforma um número float para int
3
>>> int(3.5)
3
>>> int(3.6)
3
>>> round(3.4) # Faz arredondamento de um número
3.0
>>> round(3.5)
4.0
>>> round(3.6)
4.0
>>> float(12)  # Transforma um número int para float
12.0
```

# Tipos e operações pré-definidas

- Operações relacionais

```
>>> 3 > 1 + 2         # maior
False
>>> 1 + 2 >= 2 + 1   # maior ou igual
True
>>> 4 - 1 < 4         # menor
False
>>> 4 <= 4            # menor ou igual
True
>>> 2 - 1 == 3        # igual
True
>>> 4 * 2 != 8        # diferente
False
```

# Tipos e operações pré-definidas

- Operações lógicas

```
>>> False or False    # ou
False
>>> False or True
True
>>> True or False
True
>>> True or True
True
>>> False and False   # e
False
>>> False and True
False
>>> True and False
False
>>> True and True
True
```

# Tipos e operações pré-definidas

- Operações lógicas

```
>>> not False        # não
True
>>> not True
False
>>> 3 > 2 + 2 - 1 or 4 * 4 * 4 == 4 ** 3 and not 17 == 12 + 4
True
```

# Tipos e operações pré-definidas

- O tipo `str` (string - sequência de caracteres) é usado para representar nomes, textos, etc. Uma string é especificada entre apóstrofos

```
>>> 'João da Silva'
'João da Silva'
>>> '123'           # isto é uma string, não é um número
'123'
```

# Tipos e operações pré-definidas

- Assim como podemos fazer operações com valores numéricos e booleanos, também podemos fazer operações com valores do tipo `str`

# Tipos e operações pré-definidas

```
>>> 'jose' + 'da' + 'silva'  # junção (concatenação)
'josedasilva'
>>> 3 * 'abc'                # repetição
'abcabcabc'
>>> len('Algoritmos')        # tamanho (quantidade de caracteres)
10
>>> 'Dados'.upper()          # cria uma nova string em maiúscula
'DADOS'
>>> '123' + 456              # '123' é uma string e não um número
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
>>> # int (e float) pode ser usado para converter
>>> # uma string para número
>>> int('123') + 456
579
>>> '123' + str(546)         # str converte um valor para string
'123456'
>>> int('a')                 # a string 'a' não representa um número
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'a'
```

# Tipos e operações pré-definidas

- Observe que existem 3 formas de especificar chamadas de funções
  - Pré-fixa (como funções na matemática)

    ```
    >>> len('teste')
    5
    ```

  - Como operadores (pode ser entendido como `+(3, 4)`)

    ```
    >>> 3 + 4
    7
    ```

  - Métodos (pode ser entendido como `upper('teste')`)

    ```
    >>> 'teste'.upper()
    'TESTE'
    ```

# Bibliotecas

# Bibliotecas

- As funções que podem ser usadas diretamente no Python são chamadas de embutidas (built-in)
- A maior parte das funções do Python estão em bibliotecas (módulos) e não são embutidas, estas funções (tipos, constantes, etc) precisam ser importadas antes de serem usadas
- Por exemplo, para usar a função `sin` (seno) é necessário importá-la do módulo `math`

```
>>> from math import sin
>>> sin(3.14/2)
0.9999996829318346
```

- Funções podem ser importadas tanto na janela de interação como na janela de definições
- Para ver outras funções definidas no módulo `math`, ou de outro nome qualquer, execute (para sair da ajuda pressione `q`)

```
>>> help('math')
```

# Tipos e operações pré-definidas

- A biblioteca padrão do Python é extensa. Veja a documentação de alguns tipos e funções
  - Funções pré-definidas
  - Tipos pré-definidos
  - Funções matemáticas

# Receita de projeto de funções

Podemos definir novas funções combinando as funções já existentes. Vamos seguir uma receita para definir novas funções

1. Cabeçalho, contrato, propósito e corpo preliminar
2. Exemplos
3. Corpo
4. Teste

# Exemplo 1

Defina uma função que produza o dobro de um dado valor.

# Exemplo 1

- Passo 1: cabeçalho - nome da função e dos parâmetros

```
def dobro(x):
```

# Exemplo 1

- Passo 1: cabeçalho - nome da função e dos parâmetros

```
def dobro(x):
```

- Passo 1: contrato - o que a função consome e produz - tipo dos dados de entrada e saída

```
def dobro(x):
    '''
    Número -> Número
    '''
```

# Exemplo 1

- Passo 1: cabeçalho - nome da função e dos parâmetros

```
def dobro(x):
```

- Passo 1: contrato - o que a função consome e produz - tipo dos dados de entrada e saída

```
def dobro(x):
    '''
    Número -> Número
    '''
```

- Passo 1: propósito - o que a função faz

```
def dobro(x):
    '''
    Número -> Número
    Produz o dobro de x.
    '''
```

# Exemplo 1

- Passo 1: corpo preliminar - colocar um valor preliminar como resultado da função, o valor deve ser do tipo correto

```
def dobro(x):
    '''
    Número -> Número
    Produz o dobro de x.
    '''
    return 0
```

# Exemplo 1

- Passo 2: exemplos - resultado esperado para algumas entradas

```
def dobro(x):
    '''
    Número -> Número
    Produz o dobro de x.
    >>> dobro(3)
    6
    >>> dobro(4.3)
    8.6
    '''
    return 0
```

- Observe que o corpo da função ainda não esta de acordo com o propósito

# Exemplo 1

- Passo 3: corpo - baseado nos passos anteriores, definir o corpo da função

```
def dobro(x):
    '''
    Número -> Número
    Produz o dobro de x.
    >>> dobro(3)
    6
    >>> dobro(4.3)
    8.6
    '''
    return 2 * x
```

# Exemplo 1

- Passo 3: corpo - baseado nos passos anteriores, definir o corpo da função

```
def dobro(x):
    '''
    Número -> Número
    Produz o dobro de x.
    >>> dobro(3)
    6
    >>> dobro(4.3)
    8.6
    '''
    return 2 * x
```

- Passo 4: testar - testar os exemplos na janela de interações

```
>>> dobro(3)
6
>>> dobro(4.3)
8.6
```

# Exemplo 1

- Após definir uma função, podemos usá-la como qualquer outra função pré-definida

```
>>> dobro(4) + 2
10
>>> dobro(1 + dobro(abs(-7)))
30
```

# Testes automatizados

# Testes automatizados

- Podemos fazer o Python executar os testes automaticamente
- Faça o download do arquivo `aed.py` na página da disciplina e salve o arquivo no mesmo diretório dos seus arquivos `.py`
- Importe a função de teste adicionando no início do seu arquivo

```
from aed import executa_testes
```

- No final do seu arquivo chame a função de teste

```
executa_testes()
```

- Salve o arquivo e pressione `F5`

# Testes automatizados

- Resultado dos testes

```
Trying:
    dobro(3)
Expecting:
    6
ok
Trying:
    dobro(4.3)
Expecting:
    8.6
ok
1 items had no tests:
    __main__
1 items passed all tests:
    2 tests in __main__.dobro
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

# Sentenças de seleção

# Sentenças de seleção

- Uma aproximação da sintaxe da estrutura de seleção em Python é

```
if cond:
    corpo_entao
else:
    corpo_senao
```

- `cond` deve ser uma expressão do tipo booleano
- `else` é opcional
- As instruções que fazem parte do `corpo_entao` e do `corpo_senao` são aquelas que estão "dentro" (4 espaços) do `if` e do `else`

# Exemplo 2

Defina uma função que encontre o valor máximo entre 3 valores dados.

# Exemplo 2 - Passo 1: cabeçalho, contrato, propósito e corpo inicial

```
def maximo(a, b, c):
    '''
    Número, Número, Número -> Número
    Devolve o valor máximo entre a, b e c
    '''
    return 0
```

# Exemplo 2 - Passo 2: exemplos

```python
def maximo(a, b, c):
    '''
    Número, Número, Número -> Número
    Devolve o valor máximo entre a, b e c
    >>> maximo(7, 1, 2)
    7
    >>> maximo(2, 8, -2)
    8
    >>> maximo(2, 3, 4)
    4
    >>> maximo(3, 1, 3)
    3
    '''
    return 0
```

# Exemplo 2 - Passo 3: corpo

```python
def maximo(a, b, c):
    '''
    Número, Número, Número -> Número
    Devolve o valor máximo entre a, b e c
    >>> maximo(7, 1, 2)
    7
    >>> maximo(2, 8, -2)
    8
    >>> maximo(2, 3, 4)
    4
    '''
    if a > b:
        if a > c:
            return a
        else:
            return c
    else:
        if b > c:
            return b
        else:
            return c
```

# Exemplo 2 - Passo 4: teste

```
Trying:
    maximo(3, 1, 3)
Expecting:
    3
ok
1 items had no tests:
    __main__
2 items passed all tests:
    2 tests in __main__.dobro
    4 tests in __main__.maximo
6 tests in 3 items.
6 passed and 0 failed.
Test passed.
```

# Variáveis

# Variáveis

- As variáveis em Python não precisam ser declarada, basta atribuir um valor para o nome da variável

```
>>> x = 10                    # definição das variáveis
>>> y = 20
>>> x
10
>>> y
20
>>> x + y
30
>>> x = 2 * x + 2 * y         # alteração da variável x
>>> x
60
```

- Apesar de ser permitido alterar o tipo do valor armazenado em uma variável, isto não é uma boa prática de programação

```
>>> x = 10    # x armazena um inteiro
>>> x = 'abc' # x armazena uma string, isto não é uma boa prátic
```

# Sentenças de repetição

# Sentenças de repetição

- O Python tem dois tipos de sentença de repetição
- Repetição pré-testada - `while`
  - A sintaxe aproximada é:

```
while condicao:
    corpo
```

  - Repete a execução do `corpo` enquanto `condicao` for verdadeira

# Sentenças de repetição

- Repetição em uma intervalo - `for`
  - A sintaxe aproximada do `for` é:

```
for nome_var in range(inicio, fim [, passo]):
    corpo
```

  - Se `passo` não for especificado, ele é considerado 1
  - Equivalente a

```
nome_var = inicio
while nome_var < fim:
    corpo
    nome_var = nome_var + passo
```

# Exemplo 3

Dados dois número inteiro $a$ e $b$, defina uma função que some todos os número pares entre $a$ e $b$.

# Listas (arranjos)

# Listas (arranjos)

- Os arranjos em Python são chamados de lista
- São especificados entre [ e ]
- No Python, os arranjos são dinâmicos e podem aumentar e diminuir de tamanho
- Usados para armazenar uma coleção de valores no mesmo tipo
  - Notas dos alunos
  - Nomes de pessoas
  - Etc
- Cada elemento elemento é acessado por um índice (começando de 0)

# Listas (arranjos)

```
>>> xs = [1, 4, 5]  # cria uma lista com 3 elementos
>>> len(xs)         # tamanho (quantidade de elementos)
3
>>> xs[0]           # acessa o elemento na posição 0
1
>>> xs[1]           # acessa o elemento na posição 1
4
>>> xs[-1]          # acessa o elemento na última posição
5
>>> xs[1] = 8       # altera o elemento na posição 1
>>> xs
[1, 8, 5]
>>> xs.append(-2)   # adiciona -2 no final da lista
>>> xs
[1, 8, 5, -2]
>>> del(xs[2])      # remove o elemento na posição 2
>>> xs
[1, 8, -2]
>>> xs.sort()       # ordena a lista
>>> xs
[-2, 1, 8]
```

# Efeitos colaterais e referências

# Efeitos colaterais e referências

- Nos exemplos anteriores, algumas funções não devolvem nenhum valor. Porque executar uma função que não devolve nada?

# Efeitos colaterais e referências

- Nos exemplos anteriores, algumas funções não devolvem nenhum valor. Porque executar uma função que não devolve nada?

- Pelo efeito colateral que ela gera
  - Por exemplo, a chamada `xs.sort()` não retorna nenhum valor, mas produz o efeito colateral de ordenar os valores de `xs`
- Uma função tem **efeito colateral** se ela altera algum estado do programa ou faz alguma interação observável (como imprimir na tela)
- Duas questões são importantes neste contexto
  - Como escrever funções com efeitos colaterais, especificamente as que alteram seus parâmetros?
  - Quando escrever funções sem ou com efeitos colaterais?

# Efeitos colaterais e referências

- Como escrever funções com efeitos colaterais, especificamente as que alteram seus parâmetros?
  - Através de referências

# Efeitos colaterais e referências

- Quando um valor é atribuído a uma variável, a variável passa a referenciar este valor
- Quando uma variável é atribuída a outra variável, as duas passam a referenciar o mesmo valor
- Se *um componente* do valor é alterado, a alteração é refletida nas duas variáveis, isto porque as duas referenciam o mesmo valor

```
>>> xs = [1, 6, 2, 10]
>>> ys = xs          # as duas variáveis referenciam o mesmo valor
>>> xs[1] = 7        # um componte do arranjo foi alterado
>>> xs               # a alteração é vista em xs
[1, 7, 2, 10]
>>> ys               # a alteração também é vista em ys
[1, 7, 2, 10]
>>> xs = [9, 10]     # xs referencia um novo valor, ys não é alterad
>>> ys
[1, 7, 2, 10]
```

# Efeitos colaterais e referências

- Observe que quando um novo valor é atribuído a uma variável, uma nova referencia é criada, o valor referenciado anteriormente não é alterado
- Para que a alteração de um valor seja observada por duas variáveis que referenciam este valor, é necessário alterar um componente do valor

```
>>> x = 10
>>> y = x     # as duas variáveis referenciam o mesmo valor
>>> x = x + 1 # uma nova referência é criada, y permanece inalter
>>> x
11
>>> y
10
```

# Exemplo 4

Dado um arranjo de números, escreva defina função que some 1 a cada elemento do arranjo.

1. Defina uma função que devolva um novo arranjo.
2. Defina uma função que altere o arranjo existente.

# Efeitos colaterais e referências

- Quando escrever funções sem ou com efeitos colaterais?
  - Como as funções sem efeitos colaterais são mais simples de escrever, entender e testar, deve-se dar preferência a funções sem efeitos colaterais
  - Usa-se funções com efeitos colaterais para economizar memória e tempo de execução. No exemplo, houve uma economia de memória não criando o arranjo `ys`

# Exemplo 5

Dado uma coleção de números, defina uma função que verifique se existem mais números pares ou mais números ímpares na coleção.

# Classes (estruturas)

# Classes (estruturas)

- Utilizamos classes em Python para definir estruturas (registros)
- Coleção de valores acessados pelo nome
- Os elementos de uma classe são chamados de campos ou membros
- Exemplos
  - Ponto (x, y)
  - Aluno (nome, ra, curso)
  - Produto (descrição, preço)
- Diferenças em relação ao arranjos
  - Número fixo de campos (arranjos podem ter uma quantidade qualquer de valores)
  - Cada campo tem um nome (nos arranjos cada campo - componente - é acessado por um número)
  - Cada campo pode ter um tipo diferente (nos arranjos todos os valores tem o mesmo tipo)
- Antes de usar uma classe é necessário fazer uma definição para a classe

# Classes (estruturas)

- Exemplo de definição de uma classe

```
class Ponto(object):
    '''
    Representa um ponto no plano cartesiano
        x : Número - é a coordenada x
        y : Número - é a coordenada y
    '''
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

- Toda classe deve ter uma descrição do que ela representa e a descrição dos campos

# Classes (estruturas)

- Uso da classe Ponto

```
>>> p = Ponto(2, 3)
>>> p.x
2
>>> p.y
3
>>> p.x = 10
>>> p.y = p.y + 1
>>> p.x
10
>>> p.y
4
>>> p
<__main__.Ponto object at 0x7fc9ead595d0>
>>> t = Ponto(10, 4)
>>> p == t
False
```

- Dissemos que p e t são **instâncias** da classe Ponto

# Classes (estruturas)

- A classe `Ponto` é pobre
  - Informações não relevantes quando um ponto é exibido
  - Mesmo `p` e `t` tendo as mesmas coordenadas, os pontos são considerados diferentes

# Classes (estruturas)

- Vamos usar um módulo (escrito pelo professor para ser usado nesta disciplina) que torna as classes mais ricas

```
# o arquivo ead.py deve ser baixado da página da disciplina
# e salvo no mesmo diretório deste arquivo
from aed import Struct

class Ponto(Struct):
    '''
    Representa um ponto no plano cartesiano
        x : Número - é a coordenada x
        y : Número - é a coordenada y
    '''
    def __init__(self, x, y):
        self.init(x, y)
```

# Classes (estruturas)

- Uso da classe `Ponto` (melhorada)

```
>>> p = Ponto(2, 3)
>>> p.x
2
>>> p.y
3
>>> p
Ponto(2, 3)
>>> p.x = 10
>>> p.y = p.y + 1
>>> p.x
10
>>> p.y
4
>>> p
Ponto(10, 4)
>>> t = Ponto(10, 4)
>>> p == t
True
```

- Veja que um ponto é exibido na tela da mesma forma que ele é criado, além disso, pontos com coordenadas iguais são

# Classes (estruturas)

- A sintaxe para criação de classes usando o módulo aed é

```
from ead import Struct

class NomeDaClasse(Struct):
    '''
    Descrição do que a classe representa
        campo1 : Tipo - descrição
        campo2 : Tipo - descrição
        ...
    '''
    def __init__(self, campo1, campo2, ...):
        self.init(campo1, campo2, ...)
        self._campo_interno1 = valor_inicial1
        self._campo_interno2 = valor_inicial2
        ...
```

- Os campos internos (nomes começados com _) não são especificados quando uma instância da classe é criada e também não são exibidos quando a instância é exibida na tela. Usaremos campos internos em outro momento

# Convenções de nome

- Nomes das classes com cada palavra iniciado com maiúscula
- Nomes das funções e campos começam com minúscula e as palavras são separadas por _

# Exemplo 6

Defina uma função que calcule a distância cartesiana entre dois pontos.

# Exemplo 7

Defina uma função que encontre o ponto mais distante da origem de uma lista de pontos dada.