Python

Linguagens de Programação

Celso Ademar Celante Junior
Marcos Aurélio Pereira Prado

Sumário

- Introdução
- Mini-tutorial
- Aspectos teóricos
- Avaliação

Introdução

- Linguagem de alto nível e propósito geral
- Última versão: Python 3
- Criada por Guido van Rossum e lançada em 1991
- “The Zen of Python”:
  - Python prioriza redigibilidade, legibilidade e ortogonalidade

The Zen of Python

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to
break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the
temptation to guess.
There should be one-- and
```

```
preferably only one --obvious way to do
it.
Although that way may not be obvious at
first unless you're Dutch.
Now is better than never.
Although never is often better than
*right* now.
If the implementation is hard to
explain, it's a bad idea.
If the implementation is easy to
explain, it may be a good idea.
Namespaces are one honking great idea
-- let's do more of those!
```

Executando um programa

- Para executar um script em Python, basta abrir o terminal e executar o comando:

```
>>> python modulo.py
```

Por que utilizar Python?

- Hello, World

C

```
#include <stdio.h>

int main(void) {
     printf(“Hello, World\n”);

}
```

C++

```
#include <iostream>

int main() {
     std::cout << “Hello, World\n”;

}
```

Python

```
print(“Hello, World”)
```

Sintaxe

- Comentários:

```
# comentário simples

‘‘‘ comentário
multiline ’’’
```

Palavras Reservadas

- **São palavras reservadas em Python:**

```
and     as      assert     break      class      continue
 def     del     elif    else    except     exec    finally
  for     from    global     if      import     in      is
 lambda      not     or      pass    print      raise
      return     try     while      with       yield
```

Tipagem, variáveis

- Python é uma linguagem dinamicamente tipada: não há declaração explícita de tipos de variáveis
- Fortemente tipada: impede operações entre tipos incompatíveis
- Endentação obrigatória: define blocos e hierarquias
- Case-sensitive
- Restrições sobre nomes de variáveis: 1variavel, nome$

Definição de variáveis

- **O tipo é inferido automaticamente em tempo de interpretação**

```
a = 10
# a variável é amarrada ao tipo inteiro e
recebe o valor 10

c = a + 2
# c é amarrada ao tipo inteiro e recebe a
soma do valor de a com 2
```

Saída padrão

- **Usa-se a função print() para imprimir em terminal**

```
a = 10

print(a) # imprime o valor de a e quebra
automaticamente a linha

print(“Isto é um teste”)
```

Entrada padrão

- **Usa-se a função input() [raw_input em Python 2.*] para imprimir em terminal**

```
algo = input(“Digite algo: ”)

numero = int(input(“Digite um inteiro: ”)
```

Estruturas de repetição

- if
- while
- for
- goto: Python não suporta desvios incondicionais

If

- **Estrutura de desvio condicional**

```
if expressao booleana:
   codigo se True
elif expressao booleana:
   codigo se primeiro True e segundo False
else:
   codigo se ambos forem False
```

If

- **Exemplo de estrutura de desvio condicional**

```
num = int(input(“Digite um número: ”))

if num == 2:
    print(“Dois!”)
elif num == 3:
    print(“Três!”)
else:
    print(“Não é dois e nem três!”)
```

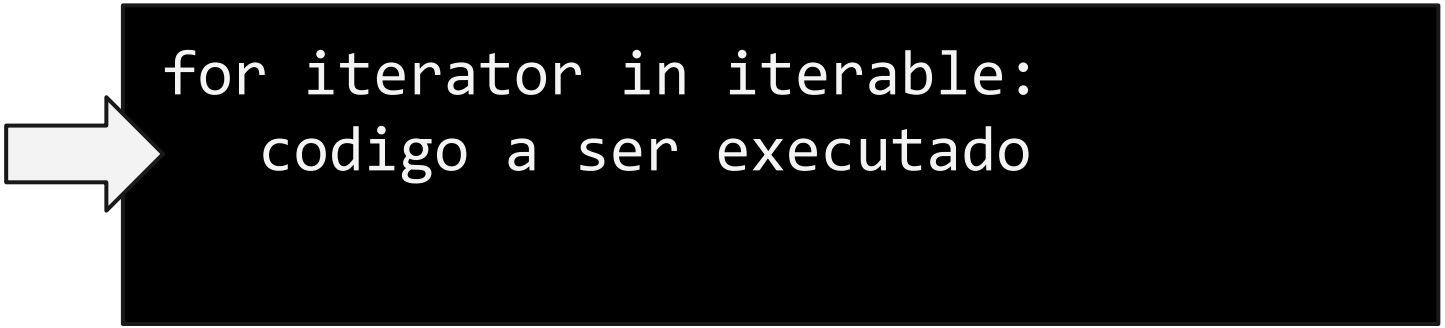
For

- **Estrutura de iteração**

```
for iterator in iterable:
   codigo a ser executado
```

For

- **Estrutura de iteração**



```
for iterator in iterable:
    codigo a ser executado
```

- **Note o uso da endentação nos exemplos acima e nos exemplos seguintes**

For

- **Exemplo de estrutura de iteração**

```
for x in range(0,3):
   print(“O número é: %d” % (x))
```

For

- **Exemplo de estrutura de iteração**

```
for x in range(0,3):
   print(“O número é: %d” % (x))
```

0, 1 e 2

range() gera um intervalo entre números, incluindo o primeiro e excluindo o último

While

- **Estrutura de repetição**

```
while expressao booleana:
   codigo a ser executado
```

While

- **Estrutura de repetição**

```
while expressao booleana:
   codigo a ser executado
```

- **Note que não se faz necessário o uso de parenteses**

While

- **Exemplo de estrutura de repetição**

```
i = 0
while i < 5:
    print(i)
    i += 1
```

Tipos de dados

- Em Python tudo é objeto: funções, tipos primitivos, tipos compostos
- Função *type(objeto)*
- Função *dir(objeto)*
- Tipos mutáveis x tipos imutáveis:

```
>>   s = “xyz”
>>   s[0] = “o”
Traceback (most recent call last):
     File “<stdin>”, line 1, in <module>
TypeError: ‘str’ object does not support item assignment
```

```
>> n = 1
>> id(n)

>> n = n + 1
>> id(n)
```

Tipos numéricos

- int: inteiro simples, 4bytes (i = 1)
- float: real de ponto flutuante (f = 3.14)
- long: dimensão arbitrária (l = 51924361L)
- complex: números complexos (c = 3.14j)

Tipos numéricos são imutáveis

Tipos numéricos

- Todos os tipos numéricos suportam operações de:
  - Adição (+)
  - Substração (-)
  - Multiplicação (*)
  - Divisão (/): inteira se realizada entre inteiros. Real caso contrário
  - Resto da divisão (%)
  - Potenciação (**)
  - Divisão inteira (//): retorna inteiro imediatamente inferior

Tipos sequenciais

- string (str)
- tupla
- lista

Strings e tuplas, como os tipos numéricos, são imutáveis, enquanto listas são mutáveis.

Strings

É possível definir uma string utilizando aspas duplas ou simples em uma única linha, ou em várias linhas com uma sequência de três aspas, duplas ou simples. São Unicode.

Operações com strings:

- Concatenação
- Interpolação
- Repetição
- Fatiamento
- Pertinência

```
>>> s = ‘string’
>>> s = ‘Uma ’ + s
>>> print “Interpolando %s de tamanho %d” % (s, len(s))
>>> s = 2*s
>>> u = u“Oi 正體字/繁”        # Unicode
>>> print s[0:]
>>> print s[3:10]
>>> ‘o’ in s
>>> for letra in s:
>>>       print letra
```

Listas

- Listas em Python são heterogêneas
- Têm suporte a fatiamento
- Podem ser concatenadas
- São iteráveis
- Vasto conjunto de métodos para manipulação:

```
>>> uma_lista = [‘a’, 2, [1]]
>>> uma_lista[0]
‘a’
>>> uma_lista.append(‘b’)
>>> uma_lista[0] = ‘A’
>>> uma_lista[0:2]
```

- append(), remove(), sort(), reverse(), etc

Tuplas

- Tuplas, bem como listas, são heterogêneas
- Têm suporte a fatiamento
- Podem ser concatenadas
- São iteráveis
- Contudo, diferentemente de listas, tuplas são imutáveis

```
>>> uma_tupla= (‘a’, 2, [1])
>>> uma_tupla[0]
‘a’
>>> uma_tupla.append(‘b’)
>>> uma_tupla[0] = ‘A’
>>> uma_tupla[0:2]
```

Tuplas

- Tuplas, bem como listas, são heterogêneas
- Têm suporte a fatiamento
- Podem ser concatenadas
- São iteráveis
- Contudo, diferentemente de listas, tuplas são imutáveis

```
>>> uma_tupla= (‘a’, 2, [1])
>>> uma_tupla[0]
‘a’
>>> uma_tupla.append(‘b’)
>>> uma_tupla[0] = ‘A’
>>> uma_tupla[0:2]
```

Não existe método append para tuplas

Tuplas

- Tuplas, bem como listas, são heterogêneas
- Têm suporte a fatiamento
- Podem ser concatenadas
- São iteráveis
- Contudo, diferentemente de listas, tuplas são imutáveis

```
>>> uma_tupla= (‘a’, 2, [1])
>>> uma_tupla[0]
‘a’
>>> uma_tupla.append(‘b’)
>>> uma_tupla[0] = ‘A’
>>> uma_tupla[0:2]
```

Não existe método append para tuplas

TypeError: object doesn’t support item assignment

Dicionários

- Um dicionário em Python é um mapeamento entre pares de valores.
- Lista de chaves e valores correspondentes
- Chaves só podem ser de tipos imutáveis (tipos numéricos, strings, tuplas)
- Já valores associados podem ser tanto mutáveis quanto imutáveis
- Dicionários são mutáveis (pode-se trocar o valor associado a uma determinada chave)

Dicionários

O acesso a itens do dicionário é feito utilizando colchetes (como em tuplas e listas) e as chaves do dicionário:

```
>>> d1 = {‘a’: 1, ‘b’: 2, ‘c’: 3}
>>> d1[‘a’]
1
>>> d2= {1: (aluno, matricula), 2: (curso, departamento)}
>>> d2[2]
(curso,departamento)
>>> d1[1]
Traceback (most recent call last):
     File “<stdin>”, line 1, in <module>
KeyError: 1
>>>
```

Tipo booleano

- Valores booleanos: True, False
- O tipo booleano é uma especialização do tipo inteiro. Assim, True equivale a 1, e False a 0
- São considerados False qualquer estrutura de dado vazia:
  - (), [], {}, ‘’

Arquivos: I/O

```
arquivo = open(“nome do arquivo”, “modo de leitura”)
```

A variável arquivo faz referência a um objeto arquivo

Arquivos: I/O

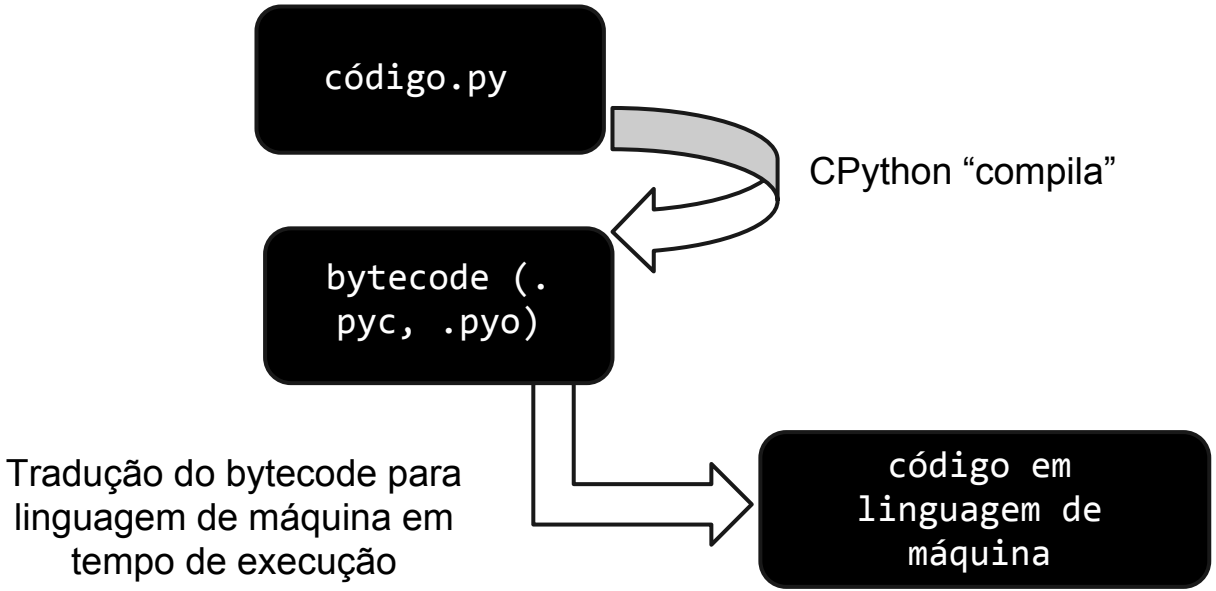
- Modos: r, w, a, r+, b (rb, wb, r+b)
- Métodos:
  - read()
  - readline()
  - write(“conteudo”)
  - writelines(lista de strings)
  - close()

É possível iterar sobre o objeto arquivo utilizando um for

Implementação da Linguagem

- Implementação padrão é hibrida
- Código é “compilado” para bytecode (.pyc, .pyo), interpretado e executado pela máquina virtual Python
- A máquina virtual Python é baseada em pilha
- Bytecode é multiplataforma e armazenado em disco

Implementação da Linguagem



- Amarrações que não foram feitas em tempo de projeto de LP são feitas em tempo de execução

Memória e coletor de lixo

- Python não oferece o uso de ponteiros
- O acesso direto a memória é possível apenas usando o módulo *ctypes*
  - Contudo não é recomendado em programação Python “pura”
  - Usa-se o módulo *ctypes* quando é necessário integrar o código Python com alguma aplicação em linguagem C

Memória e coletor de lixo

- O modo como Python gerencia o lixo em memória depende da implementação
  - CPython utiliza contagem de referências : O coletor de lixo mantém um registro de referências a cada objeto. Quando a quantidade de referências chega a zero, o objeto é destruído
- Módulo *gc*: funções para controle do coletor:
  - forçar coleta, modificar parâmetros e obter estatísticas de *debugging*

Paradigma OO

- Tudo em Python é objeto: o paradigma orientado a objetos está embutido na linguagem desde os tipos de dados mais básicos
- Classes são definidas utilizando a palavra chave *class*
- Todas as classes em Python herdam de *object*
- Python permite herança simples e múltipla e sobrescrita de métodos
- Suporta *aliasing*: diferentes nomes para mesmo objeto (não se aplica a tipos primitivos)

Classes: definição

- Classes são definidas utilizando a palavra chave *class*

```
class NomeDaClasse:
    <atributo 1>
    .
    .
    .
    <metodo 1>

class MinhaClasse:
    i = 12345
    def f(self):
        return “hello world”

x = MinhaClasse()
```

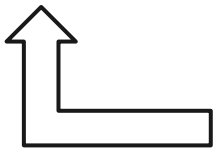
```
class UmaClasse:
    s = “abc”
    def __init__(self,at1,at2):
        self.data = []
        self.data.append(at1)
    def f(self):
        return self.data

x = UmaClasse(“LP”, 2015)
```

Classes: definição

- Classes são definidas utilizando a palavra chave *class*

```
class NomeDaClasse:
     <atributo 1>
     .
     .
     .
     <metodo 1>

class MinhaClasse:
     i = 12345
     def f(self):
          return “hello world”

x = MinhaClasse()
```

```
class UmaClasse:
     s = “abc”
     def __init__(self,at1,at2):
          self.data = []
          self.data.append(at1)
     def f(self):
          return self.data

x = UmaClasse(“LP”, 2015)
```



Método especial __init__() é chamado

Classes: definição

- Classes são definidas utilizando a palavra chave *class*

```
class NomeDaClasse:
    <atributo 1>
    .
    .
    .
    <metodo 1>

class MinhaClasse:
    i = 12345
    def f(self):
        return “hello world”

x = MinhaClasse()
```

```
class UmaClasse:
    s = “abc”
    def __init__(self,at1,at2):
        self.data = []
        self.data.append(at1)
    def f(self):
        return self.data

x = UmaClasse(“LP”, 2015)
```

Atributo estático de classe: compartilhado com todos os objetos da classe

Método especial __init__() é chamado

Classes

- Atributos ou métodos privados não existem em Python
  - Existe uma convenção de usar __ (dois *underscores)* para sinalizar que o atributo ou objeto não deve ser acessado fora da classe
- É considerado boa prática fazer comentários sobre a classe logo na primeira linha da definição. Estes podem ser acessados pelo atributo __*doc*___:

```
class MinhaClasse:
    ‘‘‘Documentação da minha classe’’’
>>> x = MinhaClasse()
>>> x.__doc__
‘Documentação da minha classe’
```

Paradigma funcional

- Python dá suporte parcial a paradigma funcional
- Uso de compreensão de listas e objetos iteráveis

```
>>> L = [1,2,3]
>>> it = iter(L)
>>> it.next()
1

for num in L
     print num

>>> palavras = “Uma frase em portugues”.split()
>>> l = [[p.upper(),p.lower(),len(p)] for p in palavras]
>>> for i in l
     print i
```

Paradigma procedural

- Python naturalmente suporta programas inteiramente procedurais, ainda que funções também sejam objetos na linguagem
- O programador pode se abster de usar os mecanismos de OO e programar de modo semelhante a programas em C, por exemplo

Funções

- Declaradas utilizando a palavra chave *def*
- Todas as funções retornam algum valor
  - Caso não tenha valor definido pelo programador, retorna *None*
- Aceita parâmetros *default* e lista de parâmetros de tamanho variado
- Aceita *docstrings*

Funções: parâmetros

- Parâmetros:
  - default
  - com nome
- Passagem:
  - cópia para tipos imutáveis
  - referência para mutáveis e objetos de classes

```
def f(nome,saudacao=‘Ola, ’,pontuacao=‘!!!’):
    ‘‘‘Docstring da
    função f’’’
    return saudacao+nome+pontuacao

>>> f(“Joao”,‘Oi’)
‘Oi Joao!!!’

>>> f(saudacao=‘Oi,’,pontuacao=‘...’,nome= “Maria”)
‘Oi,Maria...’

>>> f(pontuacao=‘...’,nome= “Jose”)
‘Ola, Jose...’
```

Funções: parâmetros

- Parâmetros:
  - lista de tamanho variado
- Passagem:
  - cópia para tipos imutáveis
  - referência para mutáveis e objetos de classes

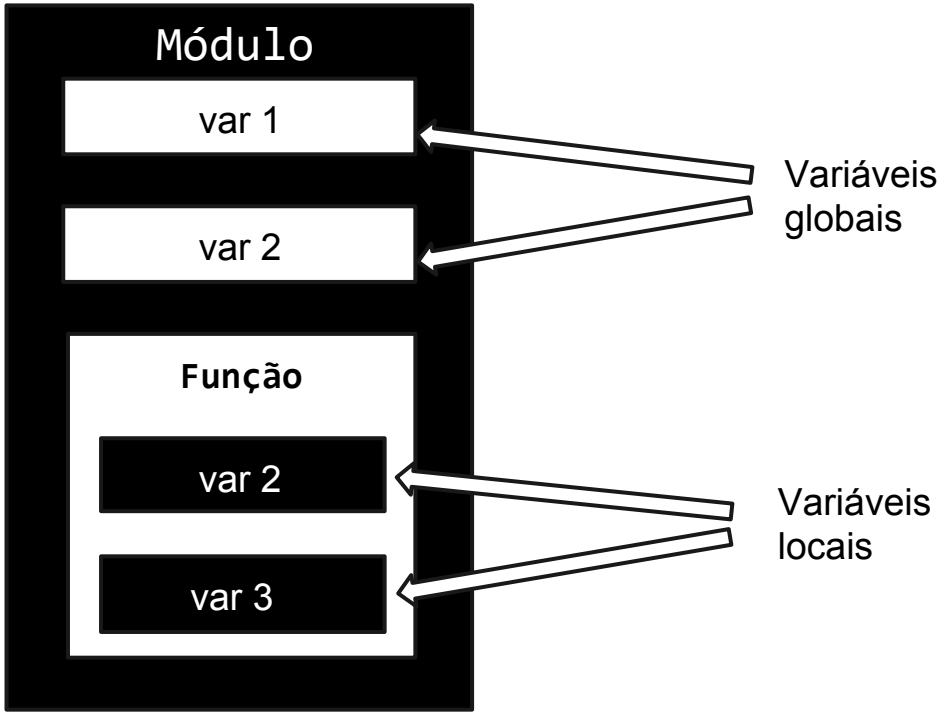
```
def f(saudacao,*nomes):
    ‘‘‘Docstring da
    função f’’’
    s=saudacao
    for nome in nomes:
        s += ‘, ’+nome
    return s+‘!’

>>> f(Ola,Joao,Maria,Jose)
‘Ola, Joao, Maria, Jose!’
```

Escopo de nomes

- Mantido através de *namespaces*:
  - Estes são mapeamentos de nomes para objetos
- *Namespaces* são implementados na maioria das vezes como dicionários
- Funções *locals()* e *globals()*: são dicionários atualizados em tempo de execução
- Escopo local é consultado antes do escopo global:
  - ofuscamento de variável

Escopo de nomes: ofuscamento

- Variáveis globais podem ser ofuscadas por variáveis locais
- Para evitar ofuscamento deve-se declarar a variável de mesmo nome como *global* no escopo local



Escopo de nomes: ofuscamento

```
def somalistas(lista):
     global soma
     for item in lista:
          if type(item) is list:
               somalistas(item)
          else:
               soma += item

soma = 0
somalistas([[1,2,10],6,[4,5]])

print soma # 28
```

Escopo de nomes: ofuscamento

Impede ofuscamento

```
def somalistas(lista):
    global soma
    for item in lista:
        if type(item) is list:
            somalistas(item)
        else:
            soma += item

soma = 0
somalistas([[1,2,10],6,[4,5]])

print soma # 28
```

Módulos

- Qualquer arquivo de código fonte em Python é um *módulo*
- Podem ser importados em outros módulos, utilizando a palavra chave *import*
- Python dá suporte:
  - importação completa do módulo (*import* modulo)
  - importação relativa (*from* arq *import* x, y)
  - apelidos para módulos (*from* arq *import* x *as* z)

Módulos: importação

```
if fibbonaci.__name__ == “__main__”:
    mensagem = ‘programa principal’

def fib1(n):
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n):
    resultado = []
    a, b = 0, 1
    while b < n:
        resultado.append(b)
        a, b = b, a+b
    return resultado
```

```
>>> import fibbonaci
>>> fibbonaci.fib1(10)
1 1 2 3 5 8

>>> from fibbonaci import fib1, fib2
>>> fib2(10)
[1, 1, 2, 3, 5, 8]

>>> import fibbonaci as fibo
>>> fibo.fib1(10)
1 1 2 3 5 8
```

Exceções

- Exceções são classes
- Exceções, exceto as do sistema, estendem a classe *Exception*
- Um procedimento não precisa sinalizar que pode lançar exceções (diferente de Java)
- Toda exceção causa encerramento do programa

Exceções: captura

- Captura exceções lançadas pelo código do block *try*
- Pode-se capturar mais informações da exceção usando *as variável*
- O *else* é opcional e só pode ocorrer depois de todos os *except*s
- *Else* é útil para informar que nenhuma exceção ocorreu

```
try:
     # código que pode lançar exceções

except ValueError:
     # tratamento de ValueError

except IOError as e:
     # tratamento de IoError

except:
     # captura qualquer outra exceção

else:
     # quando nenhuma exceção é lançada
```

Exceções: lançamento

Para lançar uma excepção, basta usar *raise*

```
class MyException(Exception):
    “”” Classe teste de exceções “”””
    def __init__(self, value):
        self.value = value
...

raise MyException
...

raise ValueError
```

Exceções: finally

- Executado logo após o bloco *try*
- Exceção lançada mas não capturada será executada logo após o *finally* ocorrer depois de todos os *except*s

```
try:
     # código que pode lançar exceções

except ValueError:
     # tratamento de ValueError

except:
     # captura qualquer outra exceção

else:
     # quando nenhuma exceção é lançada

finally:
     # código sempre executado
```

Polimorfismo

- Ad-hoc:
  - Coerção:
    - Feita implicitamente pelo interpretador Python
  - Sobrecarga:
    - de subprogramas: não suportada pela linguagem
    - de operadores: Python permite sobrecarga de operadores para classes definidas pelo programador: o programador deve implementar métodos pré-definidos na classe

Polimorfismo: sobrecarga

```
class Tempo():
    __unidades = {‘h’: 60, ‘m’: 1, ‘s’: 1/60}

    def __init__(self,t,u):
        self.tempo = t
        self.unidade = u

    def converteMin(self):
        return self.tempo * Tempo.__unidades[self.unidade]

    def __add__(self,other):
        return self.converteMin() + other.converteMin()
```

Polimorfismo: sobrecarga

```
class Tempo():
    __unidades = {‘h’: 60, ‘m’: 1, ‘s’: 1/60}

    def __init__(self,t,u):
        self.tempo = t
        self.unidade = u

    def converteMin(self):
        return self.tempo * Tempo.__unidades[self.unidade]

    def __add__(self,other):
        return self.converteMin() + other.converteMin()
```

Sobrecarga do operador binário “+”

Polimorfismo: sobrecarga

```
>>> x = Tempo(2,‘h’)
>>> y = Tempo(60,‘m’)
>>> x+y
180
>>> z = Tempo(60,‘s’)
>>> x+y+z
181
```

Polimorfismo

- Universal
  - Parâmetrico: embutido na linguagem
    - Qualquer objeto pode ser passado como parâmetro para a função ou classe
    - Uma exceção é lançada caso a implementação do subprograma tente acessar atributos ou métodos que não pertencem ao objeto passado como parâmetro

Polimorfismo

- Universal
  - Inclusão: característico da linguagem, por dar suporte a orientação a objetos
    - Python suporta herança simples e múltipla de classes

Polimorfismo: herança simples

```
class Pessoa():
    def __init__(self,n):
        self.nome = n

class Aluno(Pessoa):
    def __init__(self,n,e=‘’):
        self.escola = e
        Pessoa.__init__(self,n)

class Professor(Pessoa):
    def __init__(self,n,e,f):
        self.escola = e
        self.formacao = f
        Pessoa.__init__(self,n)
```

Polimorfismo: herança múltipla

```
class Azul():
     cor = ‘azul’
     nivel = ‘1’
     tom = ‘marinho’
     def ehAzul(self):
          return True

class Verde():
     cor = ‘verde’
     nivel = 2
     def ehAzul(self):
          return False
```

```
class Teste():
     def printcor(self):
          print self.cor
          print self.nivel
          print self.tom



class TesteCor(Teste, Verde, Azul):
     def ehAzul(self):
          return ‘Nunca!’
```

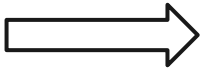
Polimorfismo: herança múltipla

```
class Azul():
     cor = ‘azul’
     nivel = ‘1’
     tom = ‘marinho’
     def ehAzul(self):
          return True

class Verde():
     cor = ‘verde’
     nivel = 2
     def ehAzul(self):
          return False
```

```
class Teste():
     def printcor(self):
          print self.cor
          print self.nivel
          print self.tom



class TesteCor(Teste, Verde, Azul):
     def ehAzul(self):
          return ‘Nunca!’
```

Polimorfismo: herança múltipla

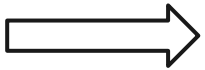
O que acontece?



```
>>> c = TesteCor()
>>> c.printcor()
```

Polimorfismo: herança múltipla

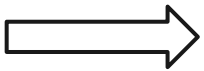
TesteCor herdou ‘cor’ e ‘nivel’ da classe Verde



```
>>> c = TesteCor()
>>> c.printcor()
verde
2
marinho
```

Polimorfismo: herança múltipla

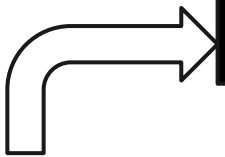
False, True
ou
‘Nunca!’?



```
>>> c = TesteCor()
>>> c.printcor()
verde
2
marinho

>>> print c.ehAzul()
```

Polimorfismo: herança múltipla

```
>>> c = TesteCor()
>>> c.printcor()
verde
2
marinho

>>> print c.ehAzul()
‘Nunca!’
```



O método ‘ehAzul’ é sobrescrito pela classe TesteCor.
Mesmo se não fosse, ‘ehAzul()’ seria False, pois TesteCor herda primeiro da classe Verde, que sobrescreve o método ‘ehAzul’ da classe Azul

Concorrência

- Suporte pela biblioteca padrão à programação com threads
- Oferece também, por padrão, ferramentas de sincronização como semáforos, monitores etc.

Concorrência: threads

- Basta criar uma classe que estenda à classe *Thread* do pacote/módulo *threading*
- Implementar o método *run()*, onde fica o código a ser executado pela thread
- Inicia-se a thread chamando o método *start()* de uma instância

Concorrência: threads

```
# tarefa.py

from threading import Thread

class Tarefa(Thread):

          def run(self):
               # código a ser executado pela thread
               i = 0
               while i < 5:
                    print(i)
                    i += 1

tarefa1 = Tarefa()
tarefa1.start()
```

Concorrência: threads

```
>> python tarefa.py
0
1
2
3
4
```

Concorrência: semáforos

- Basta criar um objeto da classe *Semaphore*, do módulo *threading*
- Se nenhum argumento for passado, o semáforo é iniciado com valor 1
- Usa-se o método *acquire*() como o P (testa/down) e *release*() como o V (up)

Concorrência: semáforos

```
from threading import Semaphore

semaforo = Semaphore() # cria-se o semáforo com valor 1

...

semaforo.acquire()

# código a ser protegido

semaforo.release()

...
```

Concorrência: monitores

- Basta criar um objeto da classe *Lock*, do módulo *threading*
- Usa-se o objeto da classe Lock junto com a a palavra reservada *with*

Concorrência: monitores

```
from threading import Lock

lock = Lock()

with lock:
    # código a ser protegido
```

Persistência de dados

- Python oferece, por padrão, algumas formas de persistência de dados
  - Serialização: permite serializar dados em JSON ou em formato Pickle
  - Bancos de dados: APIs de integração a sistemas de bancos de dados (suporte a SQLite por padrão)

Serialização

- Suporta a serialização de objetos de qualquer classe através do módulo Pickle

```
import pickle
l1 = Livro("Ensaio sobre a Cegueira", 50)
arq1 = open("serial","w") # arquivo em modo escrita
arq2 = open("serial","r+") # arquivo em modo leitura

pickle.dump(l1,arq1)
l1 = None

l1 = pickle.load(arq2)
```

LPs: propriedades desejáveis

- Legibilidade
- Redigibilidade
- Confiabilidade
- Eficiência
- Facilidade de aprendizado
- Modificabilidade
- Reusabilidade
- Portabilidade

LPs: propriedades desejáveis

- **Legibilidade**
- Redigibilidade
- Confiabilidade
- Eficiência
- Facilidade de aprendizado
- Modificabilidade
- Reusabilidade
- Portabilidade

Legibilidade

- Sintaxe clara e concisa. Legibilidade é prioridade
- Demarcadores de blocos: não são necessários
  - Usa-se endentação
- Python não possui desvios incondicionais (goto):
  - evita código “macarrônico”
- Caracteres especiais (@, $, &, etc):
  - não possuem significado especial na linguagem (diferente de C, por exemplo)
  - Mais intuitivo: uso das palavras “and”, “in”, “or”, etc

Legibilidade

- Python é multiparadigma: possui suporte a paradigma funcional, procedural e OO
  - isso permite várias maneiras diferentes de se fazer uma única programação
  - acaba possibilitando criação de código com baixa legibilidade (a partir da mistura de paradigmas)

LPs: propriedades desejáveis

- Legibilidade
- **Redigibilidade**
- Confiabilidade
- Eficiência
- Facilidade de aprendizado
- Modificabilidade
- Reusabilidade
- Portabilidade

Redigibilidade

- Altamente redigível
- Python possui tipagem dinâmica:
  - ajuda na escrita rápida de programas
- Possui tratamento de exceções:
  - evita ter que tratar casos especiais no meio do código
- Não possui mecanismo de operações com ponteiros:
  - isso evita problemas de conflito com legibilidade de código

Redigibilidade

- Não há a necessidade de usar parênteses em estruturas de controle de fluxo (if, while, for)
- Compreensão de listas:
  - recurso que permite criação de listas com poucas linhas de código
  - altamente ligado ao paradigma funcional
  - porém pode trazer prejuízos para a legibilidade do código

LPs: propriedades desejáveis

- Legibilidade
- Redigibilidade
- **Confiabilidade**
- Eficiência
- Facilidade de aprendizado
- Modificabilidade
- Reusabilidade
- Portabilidade

Confiabilidade

- Possui mecanismo para tratamento de exceções
- Não permite aritmética de ponteiros
- Apesar de dinamicamente tipada, é fortemente tipada (não permite operações entre tipos incompatíveis)
- Coletor de lixo da memória

Confiabilidade

- Nas versões 2.* (amplamente usadas) é possível mudar os valores de True e False
  - É possível fazer False = True, por exemplo

LPs: propriedades desejáveis

- Legibilidade
- Redigibilidade
- Confiabilidade
- **Eficiência**
- Facilidade de aprendizado
- Modificabilidade
- Reusabilidade
- Portabilidade

Eficiência

- Verificação dinâmica de tipos
- Controle de índice de vetor
- Por padrão, sua implementação é híbrida

LPs: propriedades desejáveis

- Legibilidade
- Redigibilidade
- Confiabilidade
- Eficiência
- **Facilidade de aprendizado**
- Modificabilidade
- Reusabilidade
- Portabilidade

Facilidade de aprendizado

- Minimimalista: poucas palavras reservadas e número reduzido de instruções básicas
- Ortogonal: menor número possível de exceções à regras
- Vasta biblioteca padrão, assim como Java, o que permite que um programador iniciante faça muitas coisas

LPs: propriedades desejáveis

- Legibilidade
- Redigibilidade
- Confiabilidade
- Eficiência
- Facilidade de aprendizado
- **Modificabilidade**
- Reusabilidade
- Portabilidade

Modificabilidade

- Python não possui constantes. Pode-se encapsular um dado em uma função e retorná-lo sempre que possível

LPs: propriedades desejáveis

- Legibilidade
- Redigibilidade
- Confiabilidade
- Eficiência
- Facilidade de aprendizado
- Modificabilidade
- **Reusabilidade**
- Portabilidade

Reusabilidade

- A linguagem encoraja a programação por orientação a objetos
- Facilita a criação de módulos/bibliotecas que podem ser facilmente distribuídos
- Frameworks de terceiros (Django e Flask, por exemplo)

LPs: propriedades desejáveis

- Legibilidade
- Redigibilidade
- Confiabilidade
- Eficiência
- Facilidade de aprendizado
- Modificabilidade
- Reusabilidade
- **Portabilidade**

Portabilidade

- Implementação mais usada é feita em ANSI C (CPython) e permite extensões escritas em C e C++
- É open source e permite implementações em outras plataformas
  - Ex.: Jython, interpretador Python em Java. Gera bytecodes compatíveis com a JVM

Avaliação da Linguagem

- Critérios gerais:
  - Aplicabilidade
  - Confiabilidade
  - Facilidade de Aprendizado
  - Eficiência
  - Portabilidade
  - Suporte ao método de projeto
  - Evolutibilidade
  - Reusabilidade
  - Integração com outros softwares
  - Custo

Avaliação da Linguagem

- Critérios específicos:
  - Escopo
  - Expressões e comandos
  - Tipos primitivos e compostos
  - Gerenciamento de memória
  - Persistência de dados
  - Passagem de parâmetros
  - Encapsulamento e proteção
  - Sistema de tipos
  - Verificação de tipos
  - Polimorfismo
  - Exceções
  - Concorrência

Avaliação da Linguagem

Critérios gerais	C	Java	Python
Aplicabilidade	Sim	Parcial	Sim
Confiabilidade	Não	Sim	Sim
Aprendizado	Não	Não	Sim
Eficiência	Sim	Parcial	Sim

Avaliação da Linguagem

Critérios gerais	C	Java	Python
Aplicabilidade	Sim	Parcial	Sim
Confiabilidade	Nã		
Aprendizado	Nã		
Eficiência	Sim	Parcial	Sim

Python é uma linguagem de propósito geral, assim como C. Apesar de não possuir mecanismos de controle de hardware (como Java), Python dá opção do programador fazer uso de bibliotecas de funções para esse tipo de programação

Avaliação da Linguagem

Critérios gerais	C	Java	Python
Aplicabilidade	Sim	Parcial	Sim
Confiabilidade	Não	Sim	Sim
Aprendizado	Não		
Eficiência	Sim		

Python, como Java, possui mecanismo de tratamento de exceções e funcionalidades que tiram do programador a responsabilidade de tratamento (coleta de lixo, verificação de índice de vetor, etc). C deixa por conta do programador todos esses tratamentos, que, se falhos, acarretam em prejuízos diversos.

Avaliação da Linguagem

Critérios gerais	C	Java	Python
Aplicabilidade	Sim		
Confiabilidade	Não		
Aprendizado	Não	Não	Sim
Eficiência	Sim	Parcial	Sim

C e Java são linguagens difíceis de aprender: C dá suporte a ponteiros enquanto que Java apresenta muitos conceitos, nem sempre ortogonais. Python, apesar de reunir grande parte dos conceitos de Java e dar suporte a paradigma funcional, é uma linguagem de fácil assimilação por ser bastante legível e concisa.

Avaliação da Linguagem

Critérios gerais	C	Java	Python
Aplicabilidade	Sim	Parcial	Sim
Confiabilidade	Não		
Aprendizado	Não		
Eficiência	Sim	Parcial	Parcial

C permite controle mais refinado dos recursos da aplicação (hardware, por exemplo). Nesse aspecto Python é semelhante a Java: as duas linguagens adicionam mecanismos de controle sobre recursos que acaba diminuindo a eficiência (verificação de tipos, coleta de lixo, etc)

Avaliação da Linguagem

Critérios gerais	C	Java	Python
Portabilidade	Não	Sim	Sim
Método de projeto	Estruturado	OO	OO e Estruturado
Evolutibilidade	Não	Sim	Sim
Reusabilidade	Sim	Sim	Sim

Avaliação da Linguagem

Critérios gerais	C	Java	Python
Portabilidade	Não	Sim	Sim
Método de projeto			
Evolutibilidade			
Reusabilidade	Sim	Sim	Sim

C possui compiladores diferentes com características diferentes, gerando uma dependência significativa. Já Python e Java tem o intuito de ser multiplataforma: portabilidade é um conceito central na linguagem

Avaliação da Linguagem

Critérios gerais	C	Java	Python
Portabilidade	Depende do paradigma escolhido para desenvolvimento do projeto. Python leva vantagem por oferecer suporte a ambos		
Método de projeto	Estruturado	OO	OO e Estruturado
Evolutibilidade	Não	Sim	Sim
Reusabilidade	Sim	Sim	Sim

Avaliação da Linguagem

Critérios gerais	C	Java	Python
Portabilidade			
Método de projeto			
Evolutibilidade	Não	Sim	Sim
Reusabilidade	Sim	Sim	Sim

C possui diversos mecanismos que podem tornar o código ilegível e difícil de manter. Já Python e Java estimulam o uso de documentação na construção de código e dão suporte ao paradigma OO, que têm evolutibilidade como ideia central

Avaliação da Linguagem

Critérios gerais	C	Java	Python
Portabilidade	Não	Sim	Sim
Método de projeto			
Evolutibilidade			
Reusabilidade	Parcial	Sim	Sim

C só dá suporte a reuso de funções. Já Python e Java dão suporte a classes e possuem mecanismos de pacotes (reuso de módulos em Python). O polimorfismo universal também auxilia na reusabilidade do código a medida que oferece mecanismos de herança e tipos genéricos.

Avaliação da Linguagem

Critérios gerais	C	Java	Python
Integração	Sim	Parcial	Sim
Custo	Depende da aplicação	Depende da ferramenta	Depende da ferramenta

Avaliação da Linguagem

Critérios gerais	C	Java	Python
Integração	Sim	Parcial	Sim
Custo			a

C e Python dão suporte a integração com diversas LPs. Já Java dá suporte apenas a integração com C/C++, necessitando de uso de uma ferramenta externa.

Avaliação da Linguagem

Critérios gerais	C	Java	Python
Integração			
Custo	Depende da aplicação	Depende da ferramenta	Depende da ferramenta

C e Python são de domńio público e embora Java esteja sob propriedade da Oracle, esta distribui gratuitamente a linguagem. Este critério depende essencialmente das escolhas da equipe que estará desenvolvendo o projeto.

Avaliação da Linguagem

Critérios específicos	C	Java	Python
Escopo	Sim	Sim	Sim
Expressões e comandos	Sim	Sim	Sim
Tipos primitivos	Sim	Sim	Sim
Gerenciamento de memória	Programador	Sistema	Sistema/Programador

Avaliação da Linguagem

Critérios específicos	C	Java	Python
Escopo	Sim	Sim	Sim
Expressões e comandos			
Tipos primitivos	Sim	Sim	Sim
Gerenciamento de memória	Programador	Sistema	Sistema/Programador

As três linguagens requerem que o programador defina explicitamente suas entidades, associando-as a um determinado escopo de visibilidade.

Avaliação da Linguagem

Critérios específicos	C	Java	Python
Escopo	Sim	Sim	Sim
Expressões e comandos	Sim	Sim	Sim
Tipos primitivos	S		
Gerenciamento de memória	Programador	Sistema	Sistema/Programador

Todas as três oferecem ampla variedade de expressões e comandos

Avaliação da Linguagem

Critérios específicos	C	Java	Python
Escopo	Sim	Sim	Sim
Expressões e comandos			
Tipos primitivos	Sim	Sim	Sim
Gerenciamento de memória	Programador	Sistema	Sistema/Programador

Todas oferecem tipos primitivos. Python e Java oferecem tipo booleano (em Python, uma especialização do tipo inteiro). Python ainda oferece mapeamentos como dicionários e estruturas flexíveis como listas heterogêneas.

Avaliação da Linguagem

Critérios específicos	C	Java	Python
Escopo	Sim	Sim	Sim
Expressões e comandos			
Tipos primitivos			
Gerenciamento de memória	Programador	Sistema	Sistema/Programador

C deixa a cargo do programador o gerenciamento da memória. Java utiliza coletor de lixo. Python usa um meio termo, pois ao mesmo tempo que utiliza coletor de lixo, também dá ao programador certo controle sobre a coleta. Ainda assim o controle sobre o gerenciamento da memória não é total como em C.

Avaliação da Linguagem

Critérios específicos	C	Java	Python
Persistência de dados	Biblioteca de funções	JDBC, biblioteca de classes, serialização	biblioteca de classes, serialização
Passagem de parâmetros	lista variável e por valor	lista variável, por valor e por copia de referência	lista variável, default, por valor e por cópia de referência
Encapsulamento e proteção	Parcial	Sim	Sim
Sistema de Tipos	Não	Sim	Sim

Avaliação da Linguagem

Critérios específicos	C	Java	Python
Persistência de dados	Biblioteca de funções	JDBC, biblioteca de classes, serialização	biblioteca de classes, serialização
Passagem de parâmetros			ault, ópia de
Encapsulamento e proteção			
Sistema de Tipos	Não	Sim	Sim

C oferece bibliotecas de I/O, mas não possui interface para BD. Java e Python suportam serialização, possuem bibliotecas para entrada e saída de dados e possuem padrões de interface para banco de dados (JDBC em Java, Sqlite e APIs em Python).

Avaliação da Linguagem

Critérios específicos	C	Java	Python
Persistência de dados			
Passagem de parâmetros	lista variável e por valor	lista variável, por valor e por copia de referência	lista variável, default, por valor e por cópia de referência
Encapsulamento e proteção	Parcial	Sim	Sim
Sistema de Tipos	Não	Sim	Sim

C usa apenas passagem por valor. Java utiliza passagem por valor e por cópia de referência. Em Python tipos mutáveis são passados por cópia de referência enquanto que tipos imutáveis são passados por valor. Python também aceita parâmetros default.

Avaliação da Linguagem

Critérios específicos	C	Java	Python
Persistência de dados	Biblioteca de funções	JDBC, biblioteca de classes,	biblioteca de classes,
Passagem de parâmetros			ault, ópia de
Encapsulamento e proteção	Parcial	Sim	Sim
Sistema de Tipos	Não	Sim	Sim

C utiliza TADs para simular mecanismo de encapsulamento (separando interface de implementação em arquivos diferentes). Java e Python oferecem mecanismos de classes.

Avaliação da Linguagem

Critérios específicos	C	Java	Python
Persistência de dados	Biblioteca de funções	JDBC, biblioteca de classes, serialização	biblioteca de classes, serialização
Passagem de parâmetros	lista variável e por valor	lista variável, por valor e por cópia de	lista variável, default, por valor e por cópia de
Encapsulamento e proteção			
Sistema de Tipos	Não	Sim	Sim

C oferece mecanismos que permitem violação de sistema de tipos (coerções, aritmética de ponteiros, etc). Java possui um sistema de tipos rigoroso. Python, apesar de permitir coerções impede operações entre tipos incompatíveis.

Avaliação da Linguagem

Critérios específicos	C	Java	Python
Verificação de tipos	Estática	Estática/Dinâmica	Dinâmica
Polimorfismo	Coerção e sobrecarga	Todos	Todos
Exceções	Não	Sim	Sim
Concorrência	Não (biblioteca de funções)	Sim	Sim

Avaliação da Linguagem

Critérios específicos	C	Java	Python
Verificação de tipos	Estática	Estática/Dinâmica	Dinâmica
Polimorfismo			
Exceções	Não	Sim	Sim
Concorrência	Não (biblioteca de funções)	Sim	Sim

C faz todas as verificações estaticamente. Java faz algumas verificações dinamicamente (checagem de índice de vetor, etc). Já Python só faz verificações dinamicamente.

Avaliação da Linguagem

Critérios específicos	C	Java	Python
Verificação de tipos			
Polimorfismo	Coerção e sobrecarga	Todos	Todos
Exceções	Não	Sim	Sim
Concorrência	Não (biblioteca de funções)	Sim	Sim

C não possui polimorfismo de inclusão ou paramétrico. Java e Python possuem todos os tipos, porém Java, ao contrário de Python, não permite sobrecarga de operadores.

Avaliação da Linguagem

Critérios específicos	C	Java	Python
Verificação de tipos	Estática	Estática/Dinâmica	Dinâmica
Polimorfismo			
Exceções	Não	Sim	Sim
Concorrência	Não (biblioteca de funções)	Sim	Sim

C não suporta mecanismos de exceções. O programador é obrigado a misturar tratamento de erro com código de fluxo normal de programação. Java e Python oferecem forte mecanismo de tratamento de exceções.

Avaliação da Linguagem

Critérios específicos	C	Java	Python
Verificação de tipos	Estática	Estática/Dinâmica	Dinâmica
Polimorfismo	Coerção e sobrecarga	Todos	Todos
Exceções			
Concorrência	Não (biblioteca de funções)	Sim	Sim

C não possui suporte nativo a concorrência. Já Python e Java oferecem recursos nativos para exclusão mútua (*synchronized*) e *threads*.

Referências

- https://docs.python.org/
- http://www.python-course.eu/
- http://nbviewer.ipython.org/github/ricardoduarte/python-para-desenvolvedores/tree/master/