Python

Ariel Monteiro, Bruno Frigeri, Murilo Canal, Luma Pontes.

INTRODUÇÃO

Características

Linguagem Interpretada

Orientada a Objeto

Quase fortemente tipada

Tipagem Dinâmica

Blocos definidos pela indentação

História

Criada em 1991 pelo matemático Guido van Rossum (foto ao lado)

Prioriza a legibilidade do código sobre velocidade ou expressividade

Baseada na linguagem ABC

Facilidade de aprendizado

Projetada com a filosofia de enfatizar o esforço do programador sobre o esforço computacional

Zen

Criado por Tim Peters, é um conjunto de “princípios” extremamente simples:

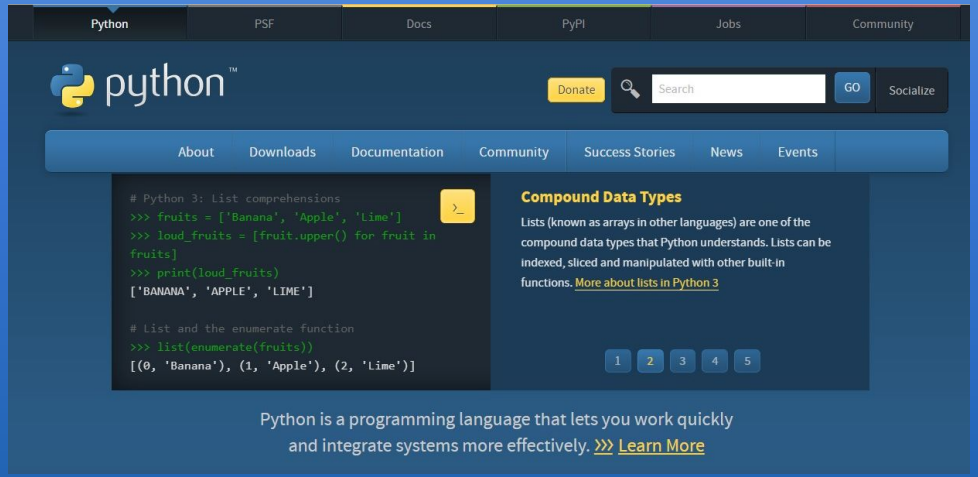
- Bonito é melhor que feio
- Explícito é melhor que implícito
- Simples é melhor que complexo

Legibilidade conta

Pode ser visualizada no código fazendo:
>> import this

```
Type "help", "copyright", "credits" or "license" for more information.
>>> import this
The Zen of Python, by Tim Peters

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!
>>>
```

Zen



python.org

Site Oficial

Versões

Python 2

- Python 2.0 - October 16, 2000
- Python 2.1 - April 17, 2001
- Python 2.2 - December 21, 2001
- Python 2.3 - July 29, 2003
- Python 2.4 - November 30, 2004
- Python 2.5 - September 19, 2006
- Python 2.6 - October 1, 2008
- Python 2.7-July 3, 2010

Python 3

- Python 3.0 - December 3, 2008
- Python 3.1 - June 27, 2009
- Python 3.2 - February 20, 2011
- Python 3.3 - September 29, 2012
- Python 3.4-March 16, 2014
- Python 3.5 - September 13, 2015
- Python 3.6- October 2016
- Python 3.7- June 2018
- Python 3.8 - October 2019

*Não é (totalmente) retrocompatível com Python 2.

Versões

Python 2

Sintaxe:

No Python 2, print não é uma função, mas sim uma declaração (***statement***). Assim, podíamos usá-la sem os parênteses

```
>>> print ‘Olá, mundo!’
```

Python 3

Sintaxe:

No Python 3, entretanto, o print() vira de fato uma ***função***, necessitando os parênteses para ser chamada.

```
>>> print(‘Olá, mundo!’)
```

Versões

Python 2

```
>>> ## Python 2
...
>>> 4/2
2
>>> 3/2
1
```

Python 3

```
>>> ## Python 3
...
>>> 4/2
2.0
>>> 3/2
1.5
>>> 4//2
2
>>> 3//2
1
```

APRENDENDO EM PYTHON

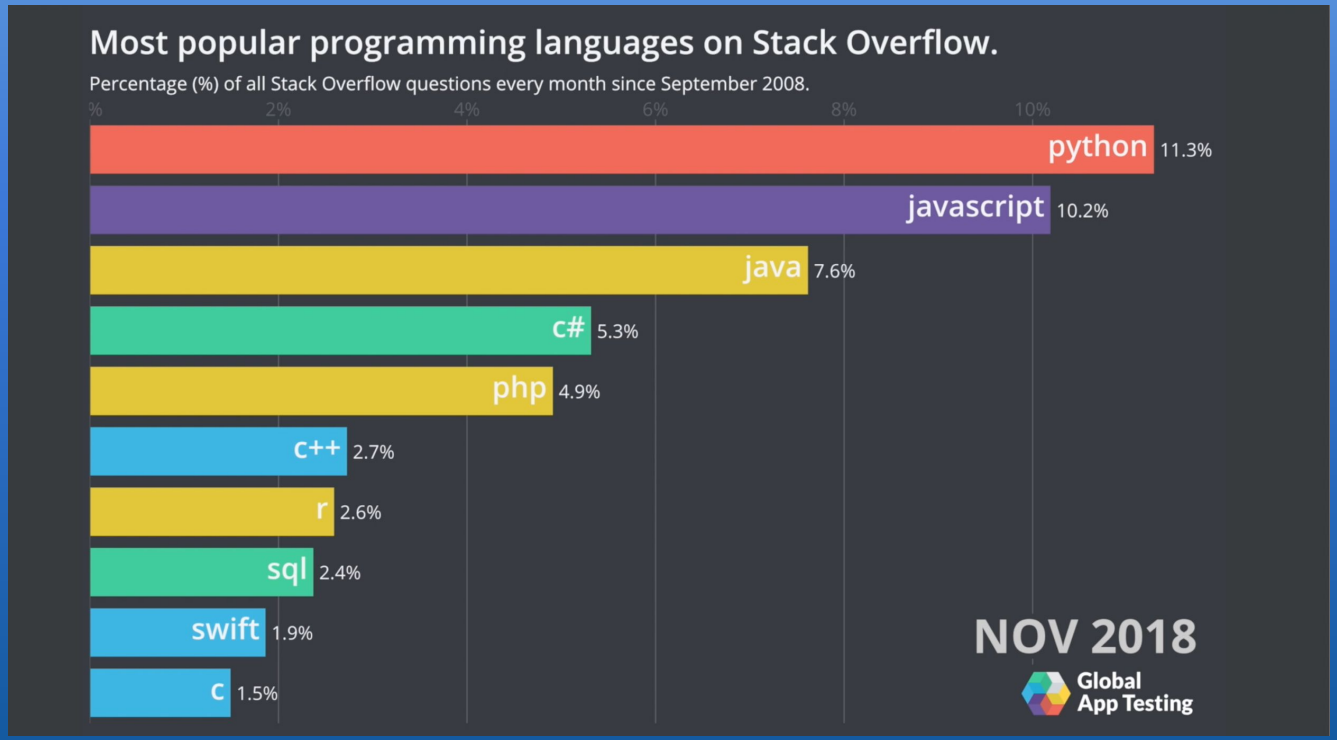
"Python has been an important part of Google since the beginning, and remains so as the system grows and evolves. Today dozens of Google engineers use Python, and we're looking for more people with skills in this language."

- Peter Norvig, director of search quality at Google, Inc.

Porque aprender Python?

Fácil de aprender | Muito usada | Versátil

Popularidade



IDEs mais usadas

- Pycharm
- Jupyter
- Visual Studio Code

Aplicações

Data Science

Machine Learning e AI

Web Development

Automatização

Implementação

Python transforma o código fonte em um executável através de um interpretador

Alguns dos vários interpretadores:

- CPython
- PyPy
- Jython

Executando um programa

Pode ser executada de duas maneiras:

- Por meio de um script
  - Executada pelo comando
    >> python script.py
- Modo interativo
  - Pode ser executado pelo terminal pelo comando >> python

Sintaxe

- Linguagem idealizada para boa legibilidade
- Utiliza palavras ao invés de símbolos para algumas operações (and e or, por exemplo)
- A indentação é utilizada para separação de blocos

Ex.:

```
if True or False:
    print("Hello World")
else:
    print("Hello World 2")
```

Amarrações

Como dito, Python segue o paradigma orientado a objeto, temos que “tudo é objeto”

Os nomes das variáveis são referências a objetos, e as suas respectivas amarrações são feitas de forma dinâmica

Referências podem ter alterações em tempo de execução, ou seja, novas amarrações são feitas

Identificadores

Case-sensitive

Os caracteres válidos para identificadores são:

- Letras maiúsculas (A _ Z) e letras minúsculas (a _ z)
- Underscore (_)
- Dígitos (0 a 9)
    - Porém, o nome do identificador não pode ser iniciado por um número

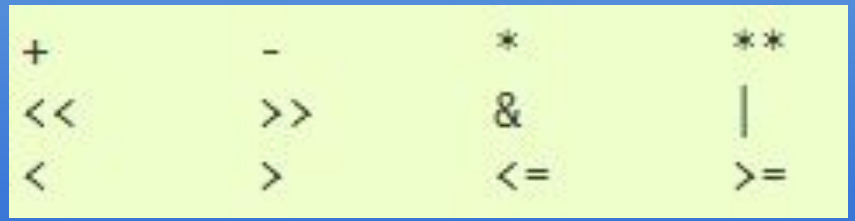
Não há limite de tamanho para identificadores

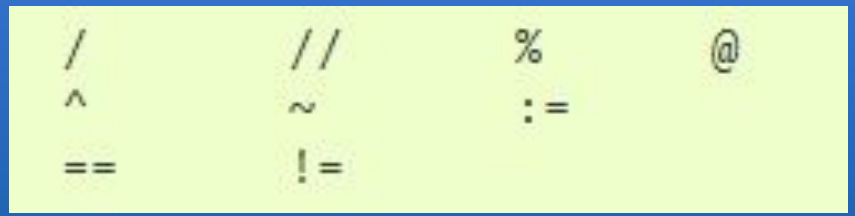
Identificadores

Palavras reservadas:

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

Operadores





PARADIGMA

Paradigma

Python segue o paradigma imperativo...

```
list_1 = ['a', 'b', 'c']
list_2 = ['a', 'b', 'c']

print(list_1 == list_2)
```

Paradigma

Python segue o paradigma imperativo...

```
list_1 = ['a', 'b', 'c']
list_2 = ['a', 'b', 'c']

print(list_1 == list_2)
```

list_1 e ***list_2*** tem valores iguais

Output:

True

... Orientado a Objeto!

```
list_1 = ['a', 'b', 'c']
list_2 = ['a', 'b', 'c']

print(list_1 is list_2)
```

is compara dois objetos

Output:

False

Paradigma

Paradigma

... Mas tem influências do paradigma funcional:

- Recursão:

```
def fatorial(n):
    if n == 1:
        return 1
    else:
        return n*fatorial(n-1)

print (fatorial(6))
```

- Lambda:

```
def adder(x):
    return lambda y: x + y

add5 = adder(5)

print(add5(1))
```

Paradigma

... Mas tem influências do paradigma funcional:

- Recursão:

```
def fatorial(n):
    if n == 1:
        return 1
    else:
        return n*fatorial(n-1)

print (fatorial(6))
```

720

- Lambda:

```
def adder(x):
    return lambda y: x + y

add5 = adder(5)

print(add5(1))
```

6

Variáveis

Declaração de tipo de variáveis não é necessário em Python

Se existir a necessidade de uma variável, apenas pense em um nome, atribua um valor e comece a usá-la

Não só o valor de uma variável pode mudar durante a execução, como seu tipo também

```
var = 2
var = 3.0
var = "var"
```

Variáveis

Pense no código em C a seguir:

```c
int x = 3;
int y = x;
y = 2;
```

O que ocorre após execução de cada linha?

Variáveis

Pense no código em C a seguir:

```c
int x = 3;
int y = x;
y = 2;
```

O que ocorre após execução de cada linha?

1. Um espaço na memória é separado para a variável **x**, nela contém o valor ***3***

Variáveis

Pense no código em C a seguir:

```c
int x = 3;
int y = x;
y = 2;
```

O que ocorre após execução de cada linha?

1. Um espaço na memória é separado para a variável ***x***, nela contém o valor ***3***
2. Um outro espaço na memória é separado para a variável ***y***, nela contém o valor de ***x*** (***3***)

Variáveis

Pense no código em C a seguir:

```c
int x = 3;
int y = x;
y = 2;
```

O que ocorre após execução de cada linha?

1. Um espaço na memória é separado para a variável ***x***, nela contém o valor ***3***
2. Um outro espaço na memória é separado para a variável ***y***, nela contém o valor de ***x*** (***3***)
3. O valor contido no espaço de memória da variável ***y*** é alterado para ***2***

Variáveis

... Mas e em Python?

```
x = 3
y = x
y = 2
```

O que ocorre após execução de cada linha?

Variáveis

... Mas e em Python?

```
x = 3
y = x
y = 2
```

O que ocorre após execução de cada linha?

1. É escolhido um local na memória para ***x*** e atribui-se a ele o inteiro 3

Variáveis

... Mas e em Python?

```
x = 3
y = x
y = 2
```

O que ocorre após execução de cada linha?

1. É escolhido um local na memória para ***x*** e atribui-se a ele o inteiro 3
2. Define-se ***y*** apontando para a mesma posição de memória que ***x***

Variáveis

... Mas e em Python?

```
x = 3
y = x
y = 2
```

O que ocorre após execução de cada linha?

1. É escolhido um local na memória para ***x*** e atribui-se a ele o inteiro 3
2. Define-se ***y*** apontando para a mesma posição de memória que ***x***
3. É escolhido um outro local na memória para ***y*** e atribui-se a ele o inteiro 2

Variáveis

Seguindo este raciocínio...

```
x = 42
x = x + 1
```

O que ocorre após execução deste código?

Variáveis

Seguindo este raciocínio...

```
x = 42
x = x + 1
```

O que ocorre após execução deste código?

Nada aponta para o endereço contendo o valor ***42***

Entramos no conceito de coleta de lixo

Coletor de Lixo

Python tem coletores de lixo diferentes para cada implementação

Mas como é o coletor em CPython?

Coletor de Lixo

Python tem coletores de lixo diferentes para cada implementação

Mas como é o coletor em CPython?

CPython faz a coleta de lixo por contagem de referência:

- Cada nó do monte contém um contador de referência atualizado em certos casos;
- Coletado quando contador = 0;

Coletor de Lixo

Python tem coletores de lixo diferentes para cada implementação

Mas como é o coletor em CPython?

CPython faz a coleta de lixo por contagem de referência:

- Cada nó do monte contém um contador de referência atualizado em certos casos;
- Coletado quando contador = 0;

Exemplo anterior:

Coletor de Lixo

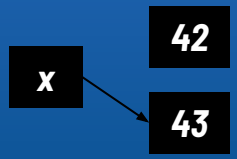
Python tem coletores de lixo diferentes para cada implementação

Mas como é o coletor em CPython?

CPython faz a coleta de lixo por contagem de referência:

- Cada nó do monte contém um contador de referência atualizado em certos casos;
- Coletado quando contador = 0;

Exemplo anterior:



Coletor de Lixo

Python tem coletores de lixo diferentes para cada implementação

Mas como é o coletor em CPython?

CPython faz a coleta de lixo por contagem de referência:

- Cada nó do monte contém um contador de referência atualizado em certos casos;
- Coletado quando contador = 0;

Exemplo anterior:

ESCOPOS E AMARRAÇÕES

Escopo

Escopo estático

- Quando um nome é utilizado em um bloco, a amarração utilizada é a definida mais próxima
- Mantidos por namespaces

Ambientes de Amarração

Como visto, Python possui blocos de amarração identificados por indentação

Variáveis declaradas em um bloco externo podem ser lidas em um bloco interno :

```
def function1():
    var = 2
    def function2():
        print(var)
    function2()
    print (var)

function1()
```

Ambientes de Amarração

Como visto, Python possui blocos de amarração identificados por indentação

Variáveis declaradas em um bloco externo podem ser lidas em um bloco interno :

```
def function1():
    var = 2
    def function2():
        print(var)
    function2()
    print (var)

function1()
```

Output:

```
2
2
```

Ambientes de Amarração

... Porém não podem ser modificadas:

```
def function1():
    var = 2
    def function2():
        var += 2
        print(var)
    function2()
    print (var)

function1()
```

Output:

ERRO

Ambientes de Amarração

Solução: ***nonlocal***

```
def function1():
    var = 2
    def function2():
        def function3():
            nonlocal var
            var += 2
            print(var)
        function3()
        print(var)
    function2()
    print (var)

function1()
```

nonlocal indica que a amarração que estamos referenciando é a definida no bloco mais próxima

Output:

```
4
4
4
```

Escopo de variáveis

Existem dois tipos de escopo para variáveis em Python, são eles:

- Local
- Global

Ex.:

```
var = "Olá"

def function():
    var = "Olá, Professor"
    print(var)

function()
print(var)
```

Escopo de variáveis

Variáveis definidas dentro de blocos são denominadas locais: possuem visibilidade somente dentro do bloco

Variáveis definidas fora de um bloco são denominadas globais: possuem visibilidade em qualquer parte do código

Ex.:

```
var = "Olá"  <- global

def function():
    var = "Olá, Professor"  <- local
    print(var)

function()
print(var)
```

Output:

Olá, Professor
Olá

Escopo de variáveis

Podemos modificar o valor da global:

Ex.:

```
var = "Olá"

def function():
    global var
    var = "Olá, Professor"
    print(var)

function()
print(var)
```

Output:

Olá, Professor
Olá, Professor

TIPOS DE DADOS

Tipos de Dados

A documentação oficial apresenta os tipos de dados da seguinte forma:

- Numéricos:
  - Inteiros;
  - Reais;
  - Booleanos;
  - Complexos;
- Sequências:
  - Imutáveis:
    - Strings;
    - Tuplas;
  - Mutáveis:
    - Listas;
- Mapeamentos:
  - Dicionário;
- I/O Object (ou file object)

Dados numéricos

- Numéricos:
  - Integer: Representam os números inteiros. Utiliza-se complemento de 2 para números negativos.
  - Boolean: Representam os valores True ou False, e podem ser representados como 1 ou 0, respectivamente.
  - Float: Representam números decimais, em python o tipo float possui precisão dupla, equivalente ao double de C.
  - Complex: Representam números complexos.

```
#int
n = 7
n = 3215464
n = 3_215_464
n = 0o177             #octal
n = 0b_1110_0101      #binário
n = 0x12ab            #hexa

#float
n = 3.14
n = 10.
n = .001
n = 3e3               #3*10^3

#complexos
n = 2j
n = 10.j
n = .01j
n = 1e100j
n = 3.123e-10j

#bool
n = True
```

Dados numéricos

Strings

- Strings: Sequência de valores codificados em Unicode que representam um texto.

```
x = "Hello World"

y = 'Hello World'
```

```
z = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
```

```
>>> s = 'teste'
>>> s
'teste'
>>> s = s + ' de strings'
>>> s
'teste de strings'
>>> len(s)
16
>>> s.upper()
'TESTE DE STRINGS'
>>> s[1]
'e'
>>> s[1:10]
'este de s'
>>> s.replace('e', 'a')
'tasta da strings'
```

Listas

- Listas são declaradas usando-se colchetes;
- Uma lista pode conter itens de tipos diferentes;
- São mutáveis.

```
>>> lista = ['morango', 1, 2+3j, True, [1, 2, 3]]
>>> lista
['morango', 1, (2+3j), True, [1, 2, 3]]
>>> lista[1] = 20
>>> lista
['morango', 20, (2+3j), True, [1, 2, 3]]
>>> len(lista)
5
>>> lista.append('LP')
>>> lista
['morango', 20, (2+3j), True, [1, 2, 3], 'LP']
>>> lista.remove(20)
>>> lista
['morango', (2+3j), True, [1, 2, 3], 'LP']
>>> lista = lista + ['a', 'b', 'c']
>>> lista
['morango', (2+3j), True, [1, 2, 3], 'LP', 'a', 'b', 'c']
>>>
```

Tipos de Dados

- Tuplas são declaradas usando-se parênteses;
- Uma tupla, assim como as listas, pode conter itens de tipos diferentes;
- Itens são indexados e NÃO podem ser modificados.

```
>>> tupla = (1, 'teste', True)
>>> tupla
(1, 'teste', True)
>>> tupla[1]
'teste'
>>> tupla[1] = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>>
>>>
```

Tipos de Dados

- Dictionary: Representa um conjunto finito de objetos arbitrários que são indexados por chaves.

- Objetos podem ser modificados

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(car)
#{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}

car["year"] = 2019
print(car)
#{'brand': 'Ford', 'model': 'Mustang', 'year': 2019}
```

File I/O

- Um *file object* é um arquivo aberto. Existem diversos modos de abrir um arquivo, mas o mais utilizado é o *open()*.
- A função *open()* tem dois parâmetros: *filename* e *mode*
  - ‘r’ - Read (é o valor default)
  - ‘a’ - Append
  - ‘w’ - Write
  - ‘x’ - Create

```
f = open('entrada.txt')
s = f.read() #lê todo o conteúdo do arquivo
f.close()

f = open('saida.txt', 'w')
f.write(s)
f.close
```

Tipagem Estática x Dinâmica

Python é dinamicamente tipada

Uma variável pode mudar de tipo ao longo da execução do programa

Ex.:

```
var = 1
print(var)
print(type(var))

var = "teste"
print(var)
print(type(var))
```

Output:

1
<class 'int'>
teste
<class 'str'>

Verificação de Tipos

O interpretador do Python avalia as expressões e não faz coerções automáticas entre tipos ***não compatíveis***.

```
i, j = 10, 20
print("O resultado é: ", i + j)
```

```
i, j = 10, "Rafael"
print(i)
print(j)
print("O resultado é: ", i + j)
```

```
10
Rafael
Traceback (most recent call last):
  File "teste", line 4, in <module>
    print("O resultado é: ", i + j)
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

OPERAÇÕES, COMANDOS E EXPRESSÕES

Operadores

Operações em Python são realizadas da direita para a esquerda

```
n = -1**2
n = (-1)**2
n = 10**-2
n = 10**(-2)
```

Operadores

Operações em Python são realizadas da direita para a esquerda

```
n = -1**2
n = (-1)**2
n = 10**-2
n = 10**(-2)
```



```
-(1²) == -1
(-1)² == 1
10-²  == 0.01
10-²  == 0.01
```

Operadores unários

“ - “ -> Inverte o sinal de um número

“ + “ -> Mantém o sinal de um número

“ ~ “ -> Inverte os bits de um número
(equivale a -(x+1))
Ex.: print(~1) -> -2

O operador ":="

Antes da versão 3.8 não era permitido fazer atribuições em expressões, como em C, por exemplo

Agora, com o operador ":=", isso é possível

```
while True:
    old = total
    total += term
    if old == total:
        return total
    term *= mx2 / (i*(i+1))
    i += 2
```

```
while total != (total := total + term):
    term *= mx2 / (i*(i+1))
    i += 2
return total
```

O comando if

```
if <expressão>:
        comando
elif <expressão>:
        comando
else:
        comando
```

```python
if nota >= 0 and nota < 7:
    print ("Prova final")
elif nota >= 7 and nota <= 10:
    print ("Aprovado")
else:
    print("Nota inválida")
```

LOOPS

while <expressão>:
    comandos

```
i = 0
while i < 10:
    print (i)
    i += 1
```

LOOPS

Usado para iterar sobre sequências

for <item> in <sequencia>:
    comandos

```python
for i in "Python":
    print(i)

grupo = ["Ariel", "Bruno", "Luma", "Murilo"]
for membro in grupo:
    print(membro)
```

```python
carro = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
for x, y in carro.items():
    print(x, y)

f = open('entrada.txt')
for line in f:
    print(line)
```

LOOPS

Em Python, os loops *while* e *for* podem ter um bloco *else*!

O bloco *else* só é chamado quando o loop termina **SEM** um *break*

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print( n, 'equals', x, '*', n/x)
            break
    else:
        # loop fell through without finding a factor
        print(n, 'is a prime number')
```

FUNÇÕES, MÓDULOS E CLASSES

Funções

Em Python, as funções são declaradas usando a palavra reservada ***def***.

Ainda, as funções também são um objeto (como visto anteriormente, expressão “tudo é objeto”).

Ex.:

```
def function(a, b):
    return a * b

print(function(2, 5))
```

Funções

Temos ainda o conceito de funções lambda em python.

Ex.:

```
funcao = lambda x, y: x + y

print(funcao(1, 2))
```

Funções

- Passagem de parâmetros pode ser posicional ou por palavras-chave

```
def function(a, b):
    return a * b

print(function(b = 2, a = 5))
```

- Suporta valores default

```
def function(a, b = 1):
    return a * b

print(function(a = 5))
```

Funções

- Direção de passagem de parâmetros unidirecional de entrada variável.
- Mecanismo de passagem por referência.
- Momento da passagem normal.
  - Avaliação do parâmetro real ocorre no momento da chamada da função.

Funções

Suporta funções com aridade variável:
*args e **kwargs

```python
def teste(arg, *args):
    print('primeiro argumento normal: {}'.format(arg))
    for arg in args:
        print('outro argumento: {}'.format(arg))

teste('python', 'é', 'muito', 'legal')

#ou
lista = ['é', 'muito', 'legal']
teste('python', *lista)

#ou
tupla = ('é', 'muito', 'legal')
teste('python', *tupla)
```

Funções

```python
def minha_funcao(**kwargs):
    for key, value in kwargs.items():
        print('{0} = {1}'.format(key, value))


minha_funcao(nome='caelum')
#nome = caelum

dicionario = {'nome': 'joao', 'idade': 25}
minha_funcao(**dicionario)
#idade = 25
#nome = joao
```

Módulos

- Um arquivo python com definição de funções e/ou classes é um módulo.
- A importação de um módulo pode ser feita utilizando a palavra-chave “import”.
- Python suporta:
  - Importação completa;
    - import arq
    - from arq import *
  - Importação relativa;
    - from arq import f,g
  - Apelidos para módulos;
    - from arq import f as funcao

Módulos

- Há de se dizer que existem diversos módulos que podem ser importados, que já vem com o compilador da linguagem, como:
    - math;
    - random;
    - os;
    - socket;

```
from math import *
x = 1
print(log(x))
```

```
import math
x = 4
print(math.sqrt(x))
```

Classes

- Em python, classes são definidas usando a palavra chave ***class***.

```
class Pessoa:
    def __init__(esta, nome, idade):
        esta.nome = nome
        esta.idade = idade
    def imprimeNome(esta):
        print(esta.nome)
    def imprimeIdade(esta):
        print(esta.idade)
```

Referência ao próprio objeto pode ser definida pelo usuário

POLIMORFISMO

Polimorfismo

Ad-Hoc

Coerção

- Feita implicitamente pelo interpretador entre tipos compatíveis.

```
i, j = 10, 2.5

print('Tipo de i:',type(i))
print('Tipo de j:',type(j))
print('Tipo de a:',type(i + j))
```

```
Tipo de i: <class 'int'>
Tipo de j: <class 'float'>
Tipo de a: <class 'float'>
```

```
i = 10
print(i)
```

```
i = 10
print(str(i))
```

Sobrecarga

- Não é suportada pela linguagem a sobrecarga de subprogramas.
- Suporta a implementação de sobrecarga de métodos especiais (ou mágicos).

```
def imprime(nome, i):
    print('O valor de', nome, 'é', i)

def imprime(i):
    print('O valor é', i)

imprime(10)
imprime('um produto', 10)
```

Sobrecarga

- Métodos Especiais
  - Uso do operador '__' , chamado dunder (**D**ouble **Under**score) como sufixo e prefixo do nome do método.
  - Normalmente usado para sobrecarga de operadores

__add__(self, other)	+
__sub__(self, other)	-
__eq__(self, other)	==

*Lista de métodos mágicos

```
class dinheiro:
    def __init__(self,valor):
        self.valor = valor
    def __str__(self):
        return 'R$' + str(self.valor)


custo = dinheiro(20.99)
print(custo)
```

```
R$ 20.99
```

Universal

Paramétrico

- Inerente da linguagem

```
def calcular(a, b, c):
    return (a+b)*c

i = calcular(1, 2, 3)
j = calcular('Para', 'métrico\n', 3)

print(i)
print(j)
```

```
9
Paramétrico
Paramétrico
Paramétrico
```

Inclusão

- Suportado pela linguagem devido a ser orientada a objeto
- Suporta herança simples e múltipla de classes

Inclusão: Herança Simples

```
class produto:
    def __init__(self, preco):
        self.preco = preco

    def ehCaro(self):
        if self.preco >= 500:
            return True
        else:
            return False
```

```
class livro(produto):
    def __init__(self, preco, pag):
        produto.__init__(self, preco)
        self.pag = pag

    def ehCaro(self):
        if self.preco >= 100:
            return True
        else:
            return False

    def ehGrande(self):
        if self.pag >= 300:
            return True
        else:
            return False
```

```
l = livro(109.99, 250)
print(l.ehCaro())
```

```
True
```

Inclusão: Herança Múltipla

```python
class produto:
    def __init__(self, preco):
        self.preco = preco

    def ehCaro(self):
        if self.preco >= 500:
            return True
        else:
            return False
```

```python
class livro():
    def __init__(self,pag):
        self.pag = pag

    def ehGrande(self):
        if self.pag >= 300:
            return True
        else:
            return False
```

```python
class hq(produto, livro):
    def __init__(self, preco, pag):
        produto.__init__(self, preco)
        livro.__init__(self, pag)


    def ehCaro(self):
        if self.preco >= 79.99:
            return True
        else:
            return False

    def ehGrande(self):
        if self.pag >= 100:
            return True
        else:
            return False
```

```python
HQ = hq(80, 150)
print(HQ.ehGrande(), HQ.ehCaro())
```

```
True True
```

EXCEÇÕES

Tratamento de Exceções

Python oferece um mecanismo próprio de tratamento de exceções.

- try / except
- try / except / finally
- try / except / else
- try / except / else / finally

try / except

```
try:
    # TENTE EXECUTAR ISSO

except:
    # SE HOUVER EXCEÇÃO, EXECUTE ISSO
```

```
try:
    # TENTE EXECUTAR ISSO

except Exce1:
    # SE HOUVER EXCEÇÃO DO TIPO Exce1,
    # EXECUTE ISSO

except Exce2:
    # SE HOUVER EXCEÇÃO DO TIPO Exce1,
    # EXECUTE ISSO

except:
    # SE HOUVER EXCEÇÃO DE OUTRO TIPO,
    # EXECUTE ISSO
```

```
try:
    with open('Arquivo.txt') as file:
        line = file.readline()
        i = int(line.strip())

except:
    print("Erro: Não foi possível ler o arquivo.")
```

```
import sys

try:
    with open('Arquivo.txt') as file:
        line = file.readline()
        i = int(line.strip())

except OSError as e:
    print("Erro:", e)

except ValueError:
    print("Erro: Dado inconsistente.")

except:
    print("Erro:", sys.exc_info()[0])
    raise
```

try / except

try / except / finally

```
try:
    # TENTE EXECUTAR ISSO

except Exce1:
    # SE HOUVER EXCEÇÃO DO TIPO Exce1,
    # EXECUTE ISSO

except:
    # SE HOUVER EXCEÇÃO DE OUTRO TIPO,
    # EXECUTE ISSO

finally:
    # ENTÃO EXECUTE ISSO
```

```
try:
    # TENTE EXECUTAR ISSO
finally:
    # ENTÃO EXECUTE ISSO
```

```
def readFile(file):
    try:
        f = open(file)
        line f.readline()
        return int(line.strip())

    except OSError:
        print("Erro ao abrir o aquivo.")
        return 0

    finally:
        print('Fim do subprograma.')
```

try / except / finally

try / except / finally

```
def bool_return():
    try:
        return True
    finally:
        return False
```

- break
- continue
- return

```
def sub():
    try:
        raise ValueError
    finally:
        return False

i = sub()
print(i)
```

```
try:
    # TENTE EXECUTAR ISSO

except Exce1:
    # SE HOUVER EXCEÇÃO DO TIPO Exce1,
    # EXECUTE ISSO

except:
    # SE HOUVER EXCEÇÃO DE OUTRO TIPO,
    # EXECUTE ISSO

else:
    # SE NÃO OCORREREM EXCEÇÕES,
    # EXECUTE ISSO
```

try / except / else

```
try:
    f = open(file, 'r')

except OSError:
    print('cannot open', file)

else:
    print(file, 'tem', len(f.readlines()), 'linhas')
    f.close()
```

try / except / else

try / except / finally / else

```python
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("Divisão por zero!")
    else:
        print("O resultado é", result)
    finally:
        print("Fim do subprograma")


divide(2, 1)
divide(2, 0)
divide('2', '1')
```

```
O resultado é 2.0
Fim do subprograma
Divisão por zero!
Fim do subprograma
Fim do subprograma
Traceback (most recent call last):
  File "C:\teste.py", line 14, in <module>
    divide('2', '1')
  File "C:\teste.py", line 3, in divide
    result = x / y
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Tratamento de Exceções

Hierarquia de classes das exceções ***built-in***:

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      |    +-- RecursionError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarnin
```

Exceções definidas pelo usuário

Deve-se criar uma classe herdeira de uma das classes de exceção.

```
class MenorIdade(Exception):
    def __init__(self):
        Exception.__init__(self,'Menor de idade')


def verifica_idade(idade):
    if int(idade) < 18:
        raise MenorIdade
    else:
        print('Idade:', idade)


verifica_idade(22)
verifica_idade(17)
```

```
Idade: 22
Traceback (most recent call last):
  File "C:\teste.py", line 15, in <module>
    verifica_idade(17)
  File "C:\teste.py", line 9, in verifica_idade
    raise MenorIdade
__main__.MenorIdade: Menor de idade
```

CONCORRÊNCIA

Tratamento de concorrência

- Python fornece mecanismo próprio para implementação de programação concorrente.
  - Módulo threading
  - São implementados também ferramentas para manipular threads com segurança, como semáforos e monitores.

Tratamento de concorrência

```python
from threading import Thread
import time
from random import random

class threadSimples(Thread):
    def run(self):
        print("Thread " + str(self.name) + ": início")
        time.sleep(random())
        print("Thread " + str(self.name) + ": fim")


threads = []

for i in range(5):
    threads.append(threadSimples(name = str(i)))
    threads[i].start()

for t in threads:
    t.join()

print('Final da execução.')
```

```
Thread 0: início
Thread 1: início
Thread 2: início
Thread 3: início
Thread 4: início
Thread 0: fim
Thread 3: fim
Thread 1: fim
Thread 2: fim
Thread 4: fim
Final da execução.
```

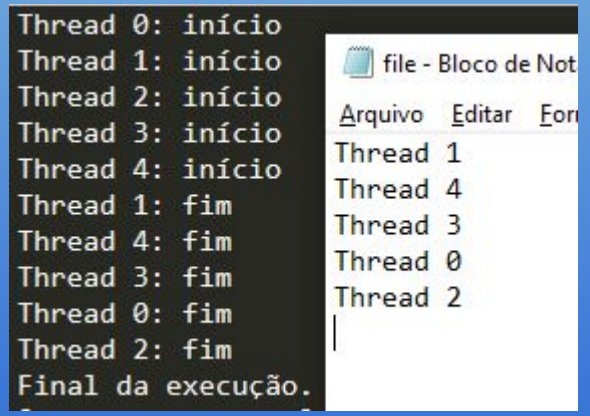
Tratamento de concorrência

```python
from threading import Thread
import time
from random import random

def threadSimples(t):
    print("Thread " + str(t) + ": início")
    time.sleep(random())
    print("Thread " + str(t) + ": fim")


threads = []

for i in range(5):
    threads.append(Thread(target=threadSimples, args=(i,)))
    threads[i].start()

for t in threads:
    t.join()

print('Final da execução.')
```

```
Thread 0: início
Thread 1: início
Thread 2: início
Thread 3: início
Thread 4: início
Thread 0: fim
Thread 2: fim
Thread 3: fim
Thread 1: fim
Thread 4: fim
Final da execução.
```

```python
from threading import Thread, Semaphore
import time
from random import random

semaforo = Semaphore()

def threadSimples(t):
    print("Thread " + str(t) + ": início")
    time.sleep(random())
    print("Thread " + str(t) + ": fim")

    semaforo.acquire()

    with open('file.txt', 'a') as f:
        f.write("Thread " + str(t) + "\n")

    semaforo.release()


threads = []

for i in range(5):
    threads.append(Thread(target=threadSimples, args=(i,)))
    threads[i].start()

for t in threads:
    t.join()

print('Final da execução.')
```

```
Thread 0: início
Thread 1: início
Thread 2: início
Thread 3: início
Thread 4: início
Thread 1: fim
Thread 4: fim
Thread 3: fim
Thread 0: fim
Thread 2: fim
Final da execução.
```



Semáforo

```
maxconnections = 5

pool_sema = BoundedSemaphore(value=maxconnections)

with pool_sema:
    conn = connectdb()
    try:
        # ... use connection ...
    finally:
        conn.close()
```

Semáforo

Monitores

```
from threading import Thread, Lock
import time
from random import random

lock = Lock()

def threadSimples(t):
    print("Thread " + str(t) + ": início")
    time.sleep(random())
    print("Thread " + str(t) + ": fim")

    with lock:
        with open('file.txt', 'a') as f:
            f.write("Thread " + str(t) + "\n")
```

```
Thread 0: início
Thread 1: início
Thread 2: início
Thread 3: início
Thread 4: início
Thread 2: fim
Thread 3: fim
Thread 0: fim
Thread 1: fim
Thread 4: fim
Final da execução.
```

file - Bloco de

Arquivo Editar

```
Thread 2
Thread 3
Thread 0
Thread 1
Thread 4
```

AVALIAÇÃO DA LINGUAGEM

Comparação entre Python, C, C++, Java

Critérios gerais	Python	C	C++	Java
Aplicabilidade	Sim	Sim	Sim	Parcial
Confiabilidade	Sim	Não	Não	Sim
Aprendizado	Sim	Não	Não	Não
Eficiência	Sim	Sim	Sim	Parcial
Portabilidade	Sim	Não	Não	Sim
Método de projeto	Estruturado, OO e funcional	Estruturado	Estruturado e OO	OO
Evolutibilidade	Sim	Não	Parcial	Sim
Reusabilidade	Sim	Parcial	Sim	Sim
Integração	Sim	Sim	Sim	Parcial
Custo	Depende da aplicação	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta

Comparação entre Python, C, C++, Java

Critérios gerais	Python	C	C++	Java
Aplicabilidade	**Sim**	**Sim**	**Sim**	**Parcial**
Confiabilidade	Sim			
Aprendizado	Sim			
Eficiência	Sim			
Portabilidade	Sim			
Método de projeto	Estruturad funcio			
Evolutibilidade	Sim	Não	Parcial	Sim
Reusabilidade	Sim	Parcial	Sim	Sim
Integração	Sim	Sim	Sim	Parcial
Custo	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta

C , C++ e Python são linguagens de propósito geral, mas 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

Comparação entre Python, C, C++, Java

Critérios gerais	Python	C	C++	Java
Aplicabilidade	Sim	Sim	Sim	Parcial
Confiabilidade	**Sim**	**Não**	**Não**	**Sim**
Aprendizado	Sim			
Eficiência	Sim			
Portabilidade	Sim			
Método de projeto	Estruturad funcio			
Evolutibilidade	Sim			
Reusabilidade	Sim			
Integração	Sim	Sim	Sim	Parcial
Custo	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta

C e C++ possuem inúmeras características estimuladoras de erros em programação (ex.: desvio incondicional irrestrito, aritmética de ponteiros, etc.). Python, assim como Java, centraliza certas operações para evitar problemas (ex.: verificação de índices de vetor, coleta de lixo, etc.)

Comparação entre Python, C, C++, Java

Critérios gerais	Python	C	C++	Java
Aplicabilidade	Sim	Sim	Sim	Parcial
Confiabilidade	Sim	Não	Não	Sim
Aprendizado	**Sim**	**Não**	**Não**	**Não**
Eficiência				
Portabilidade				
Método de projet				
Evolutibilidade				
Reusabilidade				
Integração				
Custo	ferramenta			

Embora C e Java sejam mais fáceis de aprender que C++, nenhuma dessas LPs atende ao critério. C exige uso massivo de ponteiros, que não é um conceito trivial. Java apresenta muitos conceitos, nem sempre ortogonais. Com Python, temos uma linguagem que, 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.

Comparação entre Python, C, C++, Java

Critérios gerais	Python	C	C++	Java
Aplicabilidade	Sim	Sim	Sim	Parcial
Confiabilidade	Sim	Não	Não	Sim
Aprendizado	Sim	Não	Não	Não
Eficiência	**Parcial**	**Sim**	**Sim**	**Parcial**
Portabilidade	Sim			
Método de projeto	Estruturad funcio			
Evolutibilidade	Sim			
Reusabilidade	Sim			
Integração	Sim			
Custo	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta

C e 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)

Comparação entre Python, C, C++, Java

Critérios gerais	Python	C	C++	Java
Aplicabilidade	Sim	Sim	Sim	Parcial
Confiabilidade	Sim	Não	Não	Sim
Aprendizado	Sim	Não	Não	Não
Eficiência	Sim	Sim	Sim	Parcial
Portabilidade	**Sim**	**Não**	**Não**	**Sim**
Método de projeto	Estruturad funcio			
Evolutibilidade	Sim			
Reusabilidade	Sim			
Integração	Sim			
Custo	Depend ferram			

C e C++possuem 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

Comparação entre Python, C, C++, Java

Critérios gerais	Python	C	C++	Java
Aplicabilidade	Sim	Sim	Sim	Parcial
Confiabilidade	Sim			
Aprendizado	Sim			
Eficiência	Sim			
Portabilidade	Sim			
Método de projeto	**Estruturado, OO e funcional**	**Estruturado**	**Estruturado e OO**	**OO**
Evolutibilidade	Sim	Não	Parcial	Sim
Reusabilidade	Sim	Parcial	Sim	Sim
Integração	Sim	Sim	Sim	Parcial
Custo	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta

Depende do paradigma escolhido para desenvolvimento do projeto. Python leva vantagem por oferecer suporte a ambos

Comparação entre Python, C, C++, Java

Critérios gerais	Python	C	C++	Java
Aplicabilidade	Sim	Sim	Sim	Parcial
Confiabilidade	Sim			
Aprendizado	Sim			
Eficiência	Sim			
Portabilidade	Sim			
Método de projeto	Estruturad funcio			
Evolutibilidade	**Sim**	**Não**	**Parcial**	**Sim**
Reusabilidade	Sim	Parcial	Sim	Sim
Integração	Sim	Sim	Sim	Parcial
Custo	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta

C possui diversos mecanismos que podem tornar o código ilegível e difícil de manter. C++ melhora isso com o uso de OO. 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

Comparação entre Python, C, C++, Java

Critérios gerais	Python	C	C++	Java
Aplicabilidade	Sim	Sim	Sim	Parcial
Confiabilidade	Sim	Não	Não	Sim
Aprendizado	Sim			
Eficiência	Sim			
Portabilidade	Sim			
Método de projeto	Estruturad funcio			
Evolutibilidade	Sim			
Reusabilidade	**Sim**	**Parcial**	**Sim**	**Sim**
Integração	Sim	Sim	Sim	Parcial
Custo	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta

C só dá suporte a reuso de funções. Já C++, 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 à medida que oferece mecanismos de herança e tipos genéricos.

Comparação entre Python, C, C++, Java

Critérios gerais	Python	C	C++	Java
Aplicabilidade	Sim	Sim	Sim	Parcial
Confiabilidade	Sim	Não	Não	Sim
Aprendizado	Sim	Não	Não	Não
Eficiência	Sim	Sim	Sim	Parcial
Portabilidade	Sim	Não	Não	Sim
Método de projeto	Estruturad funcio			
Evolutibilidade	Sim			
Reusabilidade	Sim			
Integração	**Sim**	**Sim**	**Sim**	**Parcial**
Custo	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta	Depende da ferramenta

C, 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.

Comparação entre Python, C, C++, Java

Critérios gerais	Python	C	C++	Java
Aplicabilidade	Sim	Sim	Sim	Parcial
Confiabilidade	Sim	Não	Não	Sim
Aprendizado	Sim	Não	Não	Não
Eficiência	Sim			
Portabilidade	Sim			
Método de projeto	Estruturad funcio			
Evolutibilidade	Sim			
Reusabilidade	Sim			
Integração	Sim			
Custo	**Depende da ferramenta**	**Depende da ferramenta**	**Depende da ferramenta**	**Depende da ferramenta**

C, C++ e Python são de domínio 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.

Comparação entre Python, C, C++, Java

Critérios gerais	Python	C	C++	Java
Escopo	Sim	Sim	Sim	Sim
Expressões e comandos	Sim	Sim	Sim	Sim
Tipos primitivos e compostos	Sim	Sim	Sim	Sim
Gerenciamento de memória	Sistema / Programador	Programador	Programador	Sistema
Passagem de parâmetros	Lista variável, default, por valor e por cópia de referência	Lista variável e por valor	Lista variável, default, por valor e por referência	Lista variável, por valor e por cópia de referência

Comparação entre Python, C, C++, Java

Critérios gerais	Python	C	C++	Java
Encapsulamento e proteção	Sim	Parcial	Sim	Sim
Sistema de tipos	Sim	Não	Parcial	Sim
Verificação de tipos	Dinâmica	Estática	Estática / Dinâmica	Estática / Dinâmica
Polimorfismo	Todos	Coerção e sobrecarga	Todos	Todos
Exceções	Sim	Não	Parcial	Sim
Concorrência	Sim	Não (biblioteca de funções)	Não (biblioteca de funções)	Sim