Seminário de Linguagens de Programação

Grupo:
Felipe de Aquino Fernandes

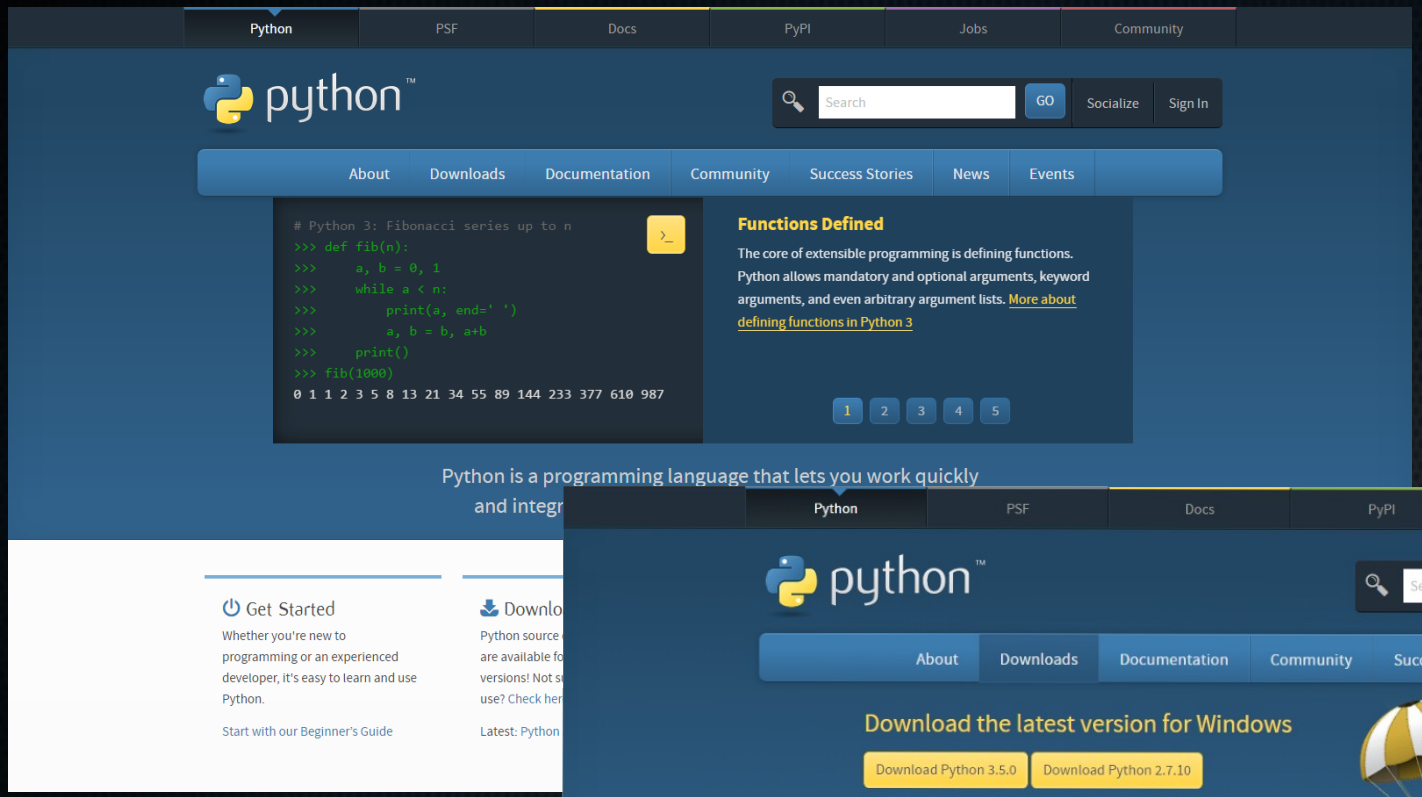
História

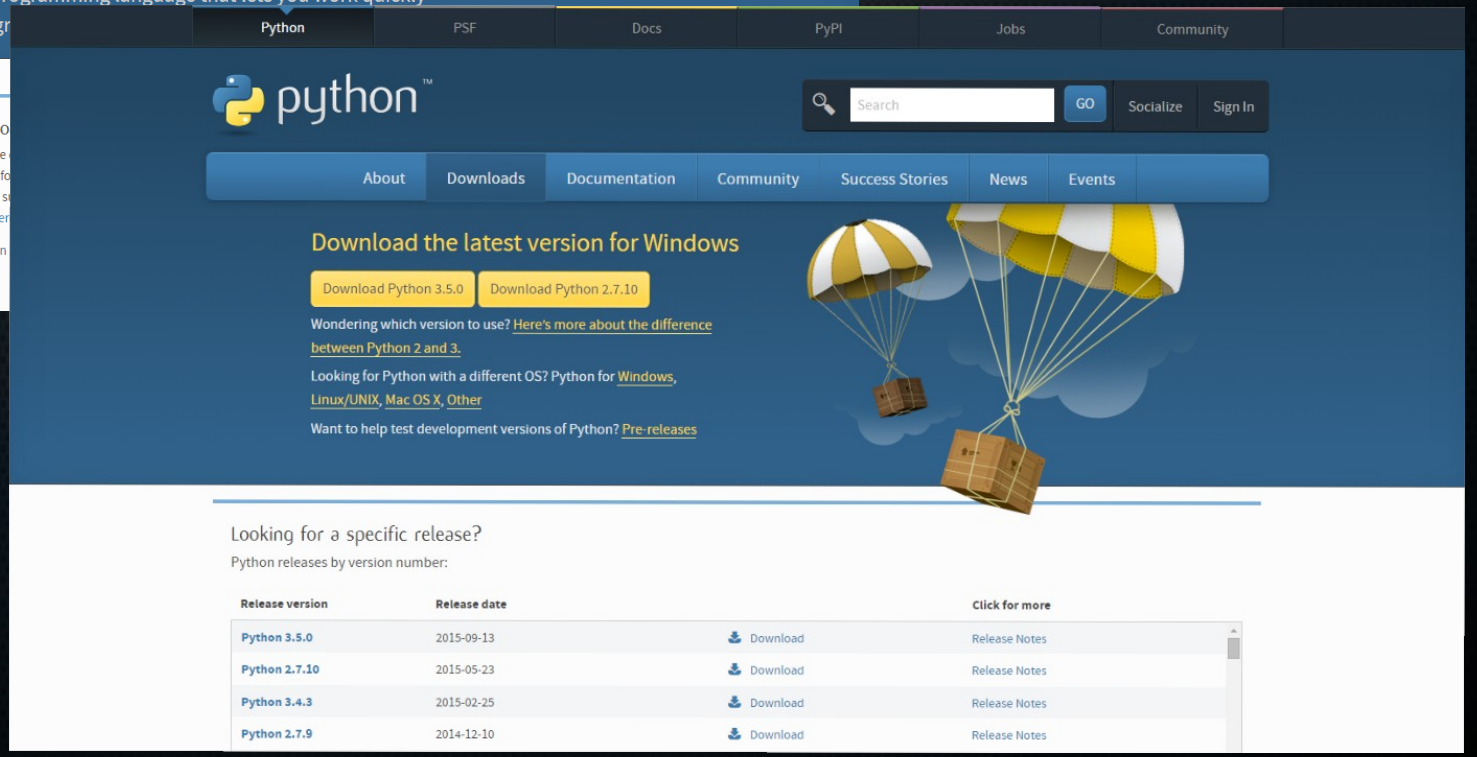
- Surgiu em 1989;
- Criada por Guido van Rossum;
- Inspiração para o nome: Monty Python and the Flying Circus;
- Licença compatível com Software Livre;
- Python foi criada para ser produtiva e fácil de escrever.

Algumas aplicações

- Computação gráfica;
- Banco de dados e mineração de dados;
- Desenvolvimento de documentação;
- Sistemas embarcados;
- Desenvolvimento de games;
- Desenvolvimento web;
- Computação científica;
- Simulações;
- Bioinformática;

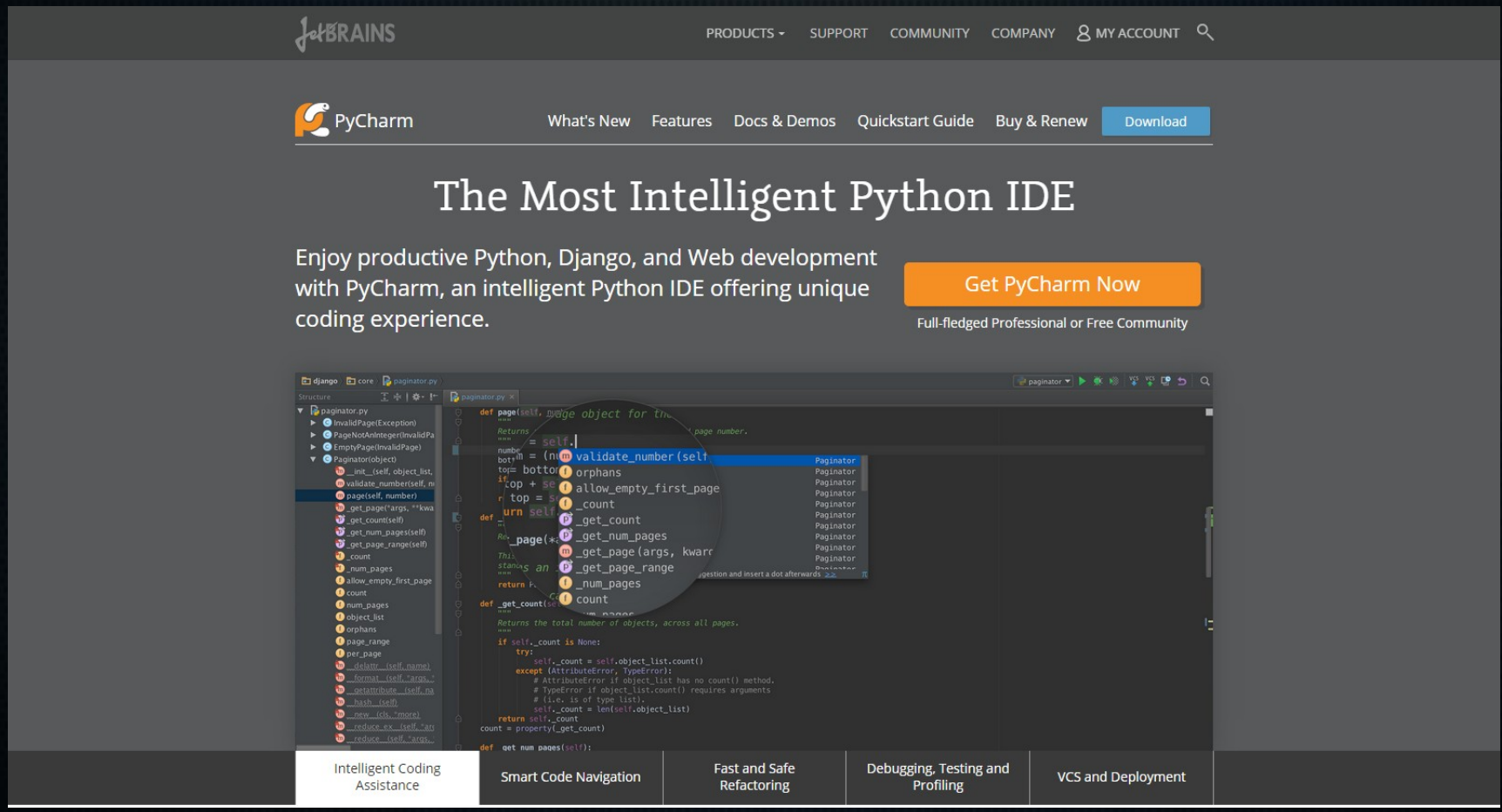
Download in www.python.org





PyCharm: Uma IDE para python

Download in https://www.jetbrains.com/pycharm/



Visão geral de Python

O site da linguagem informa:

Python é uma linguagem clara e poderosa de programação orientada a objeto.

Características notáveis:

- Usa uma sintaxe elegante, para fazer seus códigos mais fáceis de se ler;
- É uma linguagem fácil de usar;
- Vem com uma grande biblioteca padrão;
- É facilmente estendida adicionando módulos de LPs compiladas.

Alguns recursos de LPs de Python:

- Uma variedade de tipo de dados básicos estão disponíveis: numbers (floating point, complex, and unlimited-length long integers), strings (both ASCII and Unicode), lists, and dictionaries.

```
>>> print(123456789123456789123456789123456789)
123456789123456789123456789123456789
>>> (2**32) - 1
4294967295
>>> (2**64) - 1
18446744073709551615
>>> log(123456789123456789123456789123456789, 2)
116.57148949626195
```

Visão geral de Python

- Código pode ser agrupado dentro de módulos e pacotes;

```
>>> import math
>>> from math import log, exp
```

- A linguagem permite a criação e a captura de exceções;

```
>>> try:
        r = 1/0
except ZeroDivisionError as detail:
        print('Lidando com o erro em tempo de execucao: ',detail)


Lidando com o erro em tempo de execucao:  division by zero
```

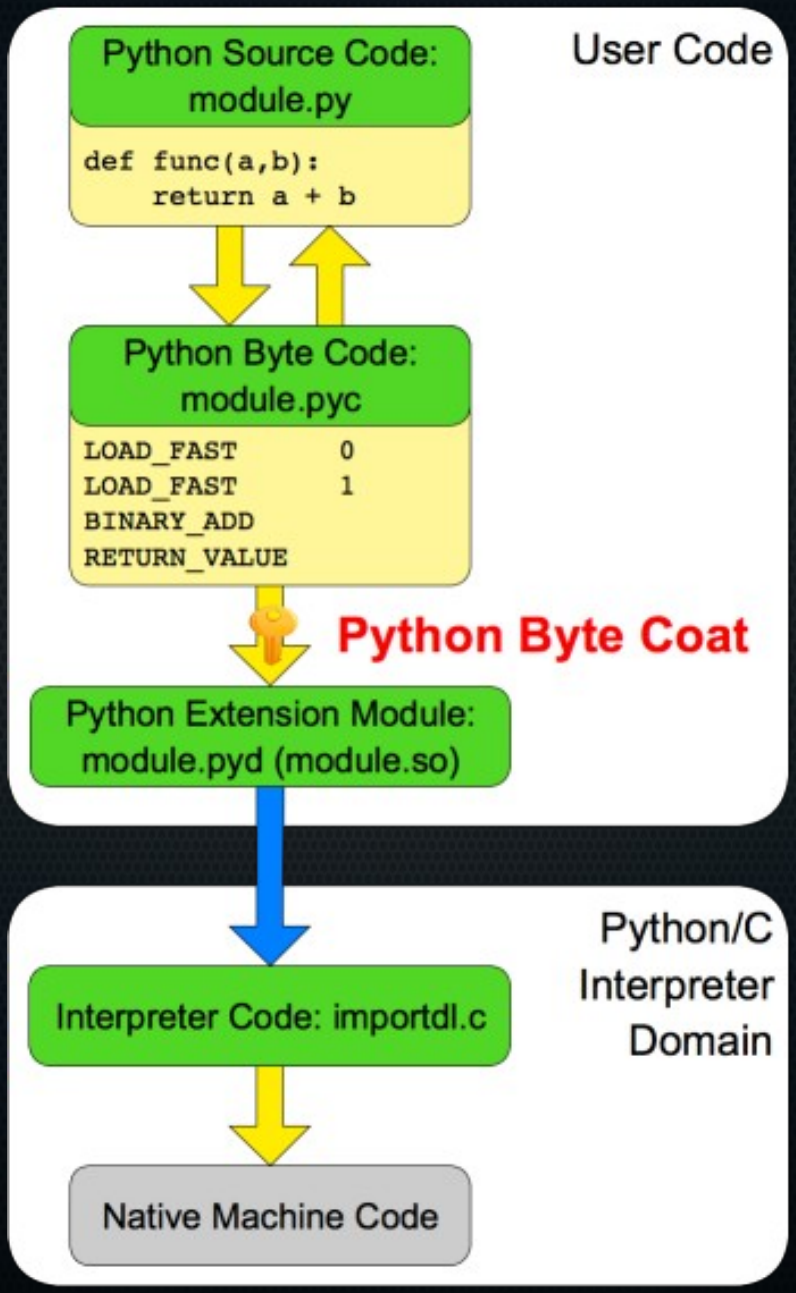
- É fortemente tipada e dinamicamente tipada;
- É uma linguagem híbrida, sendo compilada e interpretada;
- Possui gerenciamento automático de memória.
- É case sensitive
- Não tem limite para o tamanho do identificador, mas este deve inicia com uma letra

Visão geral de Python

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!
```

Visão geral de Python: Implementação



Visão geral de Python

Sintaxe: Tipos de dados e for

```
lista = [2 + 3j, 41, 3.1415926, ("int", "long int"), ["dictionary", "list"]]

dic_precos = { 'impressora': 699.00,
               'lâmpada': 5.90,
               'fogão': 599.90,
               'bicicleta': 499.00
             }
```

```
for i in range(0,len(lista)):
    print(lista[i])

for i in lista:
    print(i)
```

```
for u,v in dic_precos.items():
    print('Um(a) '+ u + ' custa '+ str(v) + ' reais\n' )
```

```
11
string
3.1415926
('int', 'long int')
['dictionary', 'list']
```

```
Um(a) bicicleta custa 499.0 reais

Um(a) impressora custa 699.0 reais

Um(a) lâmpada custa 5.9 reais

Um(a) fogão custa 599.9 reais
```

Visão geral de Python

Sintaxe: funções e ifs

```
#Ler um arquivo
def lerArq(nomearq):
    arq = open(nomearq)
    linhas =[]
    for linha in arq:
        linhas.append(linha.rstrip("\n"))
    arq.close()
    return linhas
```

```
def retornaNovaCat(categoria):
    if categoria == "prof":
        return Prof()
    elif categoria == "serv":
        return Serv()
    elif categoria == "grad":
        return Grad()
    elif categoria == "pos":
        return PosGrad()
    elif categoria == "serv_grad":
        return ServGrad()
    elif categoria == "prof_grad":
        return ProfGrad()
    elif categoria == "prof_pos":
        return ProfPos()
    elif categoria == "serv_pos":
        return ServPos()
```

Visão geral de Python

A documentação da linguagem apresenta:

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

```
+       -       *       **      /       //      %
<<      >>      &       |       ^       ~
<       >       <=      >=      ==      !=
```

```
(       )       [       ]       {       }
,       :       .       ;       @       =
+=      -=      *=      /=      //=     %=
&=      |=      ^=      >>=     <<=     **=
```

Visão geral de Python

O escopo da linguagem é estático:

```
a = 5

def f():
    b = 3
    c = a + b
    # a = 8
    c += a
    print(c)

f()
a = 3
f()
```

13
9

```
def sub():
    x = 1
    def sub1():
        print(x)

    def sub2():
        x = 3
        sub1()

    sub2()
    sub1()

sub()
```

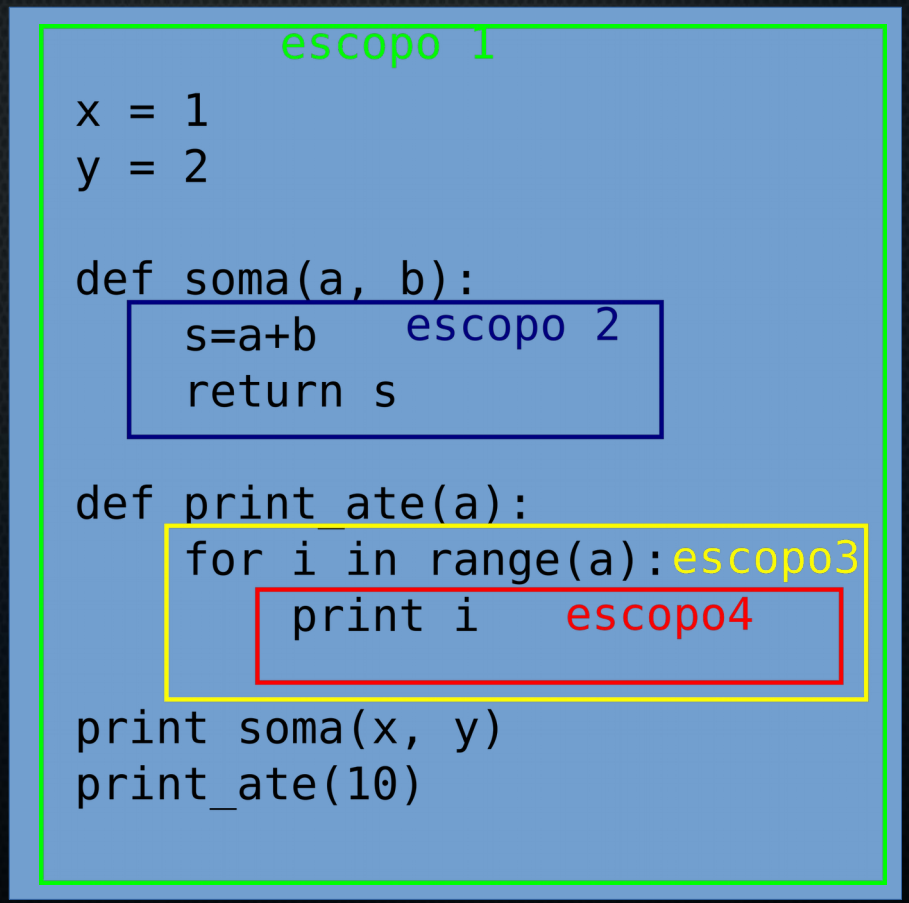
1
1

Visão geral de Python

Escopo de nomes:

- São mantidos por Namespaces, ou seja, por dicionários que relacionam os nomes dos objetos e os objetos entre si.

- Existem dois dicionários que podem ser acessados pelas funções locals() e globals().



Visão geral de Python

Ofuscamento de variável global

```
a = 5

def f():
    b = 3
    c = a + b
    global a
    a = 8
    c += a
    print(c)

f()
a = 3
f()
```

16
14

Basta declarar uma variável de mesmo nome com o identificador global

Visão geral de Python

Não há “forward reference” em python.

Não há restrições para o momento de declarações de variáveis.

Se uma variável não estiver dentro do escopo de uma função ou classe, esta existirá até o fim do programa, e poderá ser vista por qualquer função ou classe que for declarada após ela.

As declarações sequenciais e recursivas são idênticas às feitas na linguagem C.

Tipos de dados

- Python possui tipagem dinâmica;
- É (quase) fortemente tipada (todos os erros de tipos são identificados em tempo de execução ou compilação);

```
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: Can't convert 'int' object to str implicitly
```

- Tudo é objeto em python;
- Python não tem tipos primitivos;
- Há apenas “Built-in Types”:
  - Boolean;
  - Int, long, float, complex;
  - Str, unicode, list, tuple, bytearray, buffer, xrange;
  - Set, frozenset;
  - Dict;

Tipos de dados

Os tipos compostos:

- Produto cartesiano: Tuplas e classes sem métodos.
- União: Não possui
- Mapeamento: Lista e dictionary;
    - Lista e dict são dinâmicos;
    - Multidimensão: Lista de listas;
    - Mapeamento de funções;
- Conjunto Potência: Set, Frozenset;

```
A = set()
B = set()

for e in range(0,6):
    A.add(e)

for e in range(4,10):
    B.add(e)

s = A.union(B) # A U B
print(s)
s = A.difference(B) # A - B
print(s)
s = B.difference(A) # B - A
print(s)
s = A.intersection(B)
print(s)
```

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
{0, 1, 2, 3}
{8, 9, 6, 7}
{4, 5}
```

Tipo de dados

- Tipos recursivos: A recursão existe apenas para funções;
- Tipo ponteiro: Não existe em python;
- Tipo referência: em python as variáveis são do tipo referência;

```
class list2(list):
    def particiona(self, esq, dir, compara):
        pivo = self[(esq+dir)//2]
        i = esq
        j = dir
        while i <= j:
            while compara(pivo,self[i]) and i < dir: i += 1
            while compara(self[j],pivo) and j > esq: j -= 1
            if i <= j:
                self[i], self[j] = self[j], self[i]
                i += 1
                j -= 1
        return i

    def quicksort(self, esq, dir, compara):
        if esq < dir:
            i = self.particiona(esq,dir,compara)
            self.quicksort(esq,i-1,compara)
            self.quicksort(i,dir, compara)
```

Propriedades das variáveis

- Em python, assim como em java, existem apenas referências para objetos:

```
>>> l1 = [1, 8, 3, 5, -7, 5, 2, 7]
>>> l2 = l1
>>> l2.sort()
>>> print(l1); print(l2)
[-7, 1, 2, 3, 5, 5, 7, 8]
[-7, 1, 2, 3, 5, 5, 7, 8]
```

- O tipo da variável é determinado semanticamente;

- Tempo de vida:
  – Locais, globais, anônimas, alocação estática;
  – Transientes ou persistentes.

Memória Secundária

Pickle e Marshal: Módulos que permitem a serialização

- Marshal não permiter a serialização de classes definidas pelo o usuário, e não garante a portabilidade para outras versões de python

Shelve: Permite a persistência de objetos do tipo dicionário.

```
>>> import shelve
>>> arq = shelve.open('lixo')
>>> arq['site'] = 'http://www.python.org'
>>> arq['pi'] = 3.1415
>>> arq['fibo'] =[0, 1, 1, 2, 3, 5, 8, 13, 21]
>>> print arq['pi']
3.1415
>>> print arq['fibo']
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
>>> del arq['site']
```

```
>>> import shelve
>>> arq = shelve.open('lixo')
>>> for chave in arq:
        print chave, arq[chave]

fibo [0, 1, 1, 2, 3, 5, 8, 13, 21]
pi 3.1415
```

Coleta de lixo

Segundo o site Digi no link : Python Garbage Collection

Python usa dois métodos para liberar memória:

- Contagem de referência: Funciona através da contagem do número de referências a um objeto, quando este número é zero este objeto é liberado;
- Automatic Garbage Collection of Cycles: como ciclos precisam de trabalho computacional para serem encontrados, python agenda essa atividade. Esse agendamento funciona da seguinte maneira: é usado um número limite, quando a diferença entre o número de objetos alocados e o número de objetos desalocados é superior ao número limite o garbage collectior é ativado.

Garbage colletion interface: permite desabilitar o coletor, modificar a frequência de coleção, e ativar opções de debug.

Mais informações em: Garbage Collector interface

Operadores e Expressões

Aritméticos e relacionais:

```
+       -       *       **      /       //      %
<<      >>      &       |       ^       ~
<       >       <=      >=      ==      !=
```

Tipos de expressões:

- Literais: 99 99.0 'str' “str” 0x43
- Agregações: lista = [1, 2, 3] string = 'string' d = {a: 1, b: 2}
- Aritméticas: num = 23 / 6 num = 23 // 6
- Binárias: print(str(10 << 2) + ' ' + str(1 | 6))
- Condicionais: x > y x != y

Closure:

```
def obterClosure():
    def println(info):
        print(info + '\n')
    return println

closure = obterClosure()
closure('Hello')
```

Expressões

-Precedência:

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Ccomplement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'td>
^ |	Bitwise exclusive `OR' and regular `OR'
<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

Expressões e Tipos de Comandos

- Categórica:

```
for m in lis:
    if isinstance(m,Prof):
        if m.retornaRegime() == "40" and m.matrFun != 0:
            lisProf.append(m)
    if isinstance(m,PosGrad):
        if m.matrEst != 0:
            lisPos.append(m)
    if isinstance(m,Estudante):
        if m.matrEst != 0:
            lisEst.append(m)
```

Tipos de comandos

- Atribuições:
  - Simples: x = a + 2*y
  - Múltipla: a = b = 1
  - Composta: a += 3 a &= 3
  - Unária: # a++ ++a -+a

Tipos de Comandos

- Sequenciais: Conjuntos de comandos como um só

```
if x > y :
        n = 1
        n += 3
        if n < 5:
                n = 10
                m = n * 2
```

bloco de comandos

- Colaterais: Comandos que permitem processamento paralelo
  - Não existe em python.

- Condicionais

```
if x > 0:
        a = 0
 elif x == 0
        a = 1
else
        a = 2
```

Não existe switch em python

Tipos de Comandos

- Iterativos

```
f = 1
y = x
while y > 0:
    f *= y
    y-= 1
```

```
for i in range(0,len(lista)):
    print(lista[i])

for i in lista:
    print(i)
```

- Desvio irrestrito: Phython não tem goto.

- Escapes:

```
soma = 0
while True:
    n = input('Number: ')
    if n == 0:
        break
    soma += n
```

```
i = 0
s = 0
while i < 10:
    n = input('Number ')
    if n < 0: continue
    s+=n
    i+=1
```

Parâmetros

Correspondência real-formal: posicional e por palavra-chave.
Suporta valores default.

```
def juroSimples(C = 0, i = 1, t = 12):
    return (C*i*t)/100

M1 = juroSimples()
M2 = juroSimples(5000,2,24)
M3 = juroSimples(5000,2)
M4 = juroSimples(5000, t = 24)
M5 = juroSimples(i = 3,C = 4000,t = 18)
```

Número de parâmetro variável

```
def paralelo(*R):
    Req = 0
    for r in R:
        Req += (1/r)
    Req = (1/Req)
    return Req

k = 1000
Rt = paralelo(10*k, 56*k, 3.9*k)
```

Parâmetros

Direção de passagem: Unidirecional de entrada variável
Mecanismo de passagem: Cópia
Momento da passagem: Normal

TADS:

```
class Fila():
    def __init__(self, info):
        self.ultimo = No(info)
        self.primeiro = self.ultimo
        self.primeiro.prox = self.ultimo

    def insere(self, info):
        novo = No(info)
        self.ultimo.prox = novo
        self.ultimo = novo

    def remove(self):
         self.primeiro = self.primeiro.prox

    def imprime(self):
        aux = self.primeiro
        while aux:
            print(str(aux.info)+'\n')
            aux = aux.prox
```

```
class No():
    def __init__(self, info):
        self.info = info
        self.prox = None
```

Módulos

Python usa tabelas símbolos.

Uma tabela de símbolos é um dicionário de dados que cada módulo possui, onde são armazenadas todas as variáveis, funções e classes definidas neste módulo.

```
>>> dir()
  ['__builtins__', '__doc__', '__name__']

>>> import math
  ['__builtins__', '__doc__', '__name__',
  'math' ]
```

Usa-se *import* para carregar módulos(arquivos) na mesma pasta ou módulos que vieram com o compilador.

Para carregar uma função de um módulo: from math import sqrt

Pacotes

```
util/
    __init__.py
    sort.py [quicksort(), bubblesort()]
    string/
        __init__.py
        format.py [Parser, Validator]
        io.py [StringIO]
    number/
        __init__.py
        format.py [DoubleFormat, IntFormat]
```

__init__.py: são arquivos especiais e servem para que o interpretador possa identificar quais diretórios são pacotes e quais não são

Polimorfismo

Ad-hoc:

-Coerção:

- Não faz a coerção em atribuições
- Em operações, não faz a coerção implícita quando strings e tipos numéricos estão envolvidos: # a = 1 + '1'
- Conversão de tipos:

```
>>> x = 12345678912345678912345678912345678 9
>>> y = float(x)
>>> z = int(y)
>>> print(y)
1.2345678912345678e+35
>>> print(z)
123456789123456784102659645885120512
```

-Sobrecarga:

- A linguagem facilita a sobrecarga de operadores. Basta definir alguns métodos em uma classe. Por exemplo:

__add__ = +	__eq__ = ==	
__sub__ = -	__ne__ = !=	
__mul__ = *	__gt__ = >	
__div__ = /	__ge__ = >=	
__mod__ = //	__lt__ = <	
__abs__ = abs	__le__ = <=	e outros

Polimorfismo

Facilidade na sobrecarga de operadores:

```
class animal():
    def __init__(self, tipo = 'animal'):
        self.tipo = tipo

    def __add__(self, other):
        return (self.tipo +'-'+ other.tipo)

class gato(animal):
    def __init__(self):
        self.tipo = 'gato'

class rato(animal):
    def __init__(self):
        self.tipo = 'rato'

    def __add__(self, other):
        return (self.tipo + ' modificado')


cat = gato()
rat = rato()

res = cat + rat
print(res)
res = rat + cat
print(res)
```

gato-rato
rato modificado

Polimorfismo

- Universal:
  - Paramétrico:

```
def identidade(x):
    return x
```

  - Inclusão:
    - Admite tipos genéricos;
    - Possui herança múltipla;
    - Todos os métodos e atributos são públicos;
    - Tem amarração tardia;
    - Permite classes e métodos abstratos: abstract;
    - Possui metaclasses.

```
class Casulo():
    def __init__(self, elem = None):
        self.elemento = elem

    def colocar(self, elem):
        self.elemento = elem

    def retirar(self):
        return self.elemento

c = Casulo()
c.colocar('Uma String')
s = c.retirar()
c.colocar(Casulo(10))
obj = c.retirar()
print(s)
print(str(obj.retirar()))
```

Exceções: Hierarquia


```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
```

Exceções: Hierarquia

```
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      |    +-- RecursionError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarning
```

Exceções

Python usa o mecanismo try-except-else-finally.
A palavra *raise* serve para propagar exceções.

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops!  That was no valid number.  Try again...")
...
```

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

Exceções

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError as e:
...     print('My exception occurred, value:', e.value)
...
```

Concorrência

- Possui módulo nativo para programação com concorrência chamado threading;
- Em threading estão implementados ferramentas para se manipular threads, semáforos e monitores;

- Threads:
  - De maneira semelhante a Java, para usar threads, cria-se uma classe que herda da classe Thread, onde deve ser implementado o método run();
  - A classe Thread, possui os métodos start() e join();

- Semáforos:
  - No módulo threading existe uma classe chamada Semaphore;
  - O construtor tem 1 como valor default;
  - O método acquire() é equivalentente à função P() e o método release() é equivalente ao método V().

Concorrência: Ex.: Threads

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

class simpleThread(Thread):

    def run(self):
        for i in range(0,5):
            print(str(i) + ' ' + str(self.name)+'\n')
            time.sleep(int(random.random()*100)//5)
        print('Fim de ' + str(self.name))


threadA = simpleThread(name = 'TA')
threadB = simpleThread(name = 'TB')

threadA.start()
threadB.start()

try:
    threadA.join()
except:
    pass

try:
    threadB.join()
except:
    pass
```

```
0 TA
0 TB


1 TA

1 TB

2 TB

2 TA

3 TA

3 TB

4 TB

Fim de TB
4 TA

Fim de TA
```

Concorrência

Exemplo de semáforo:

```
from threading import Semaphore

semaforo = Semaphore()

# parte do código

semaforo.acquire()

# código a ser protegido(ou região crítica)

semaforo.release()

# parte do código
```

Em python existe outro tipo de semáforo: BoundedSemaphore

Concorrência

- Monitores:
  - Python possui um recurso semelhante a de monitores, apresentado a seguir:

```
from threading import Lock

lock = Lock()

with lock:
    # código a ser protegido(ou região crítica)
```

  - O uso de *with* juntamente com um objeto da classe Lock() ou Semaphore(), faz com que o código a ser protegido receba as operações acquire() e release() automaticamente.

Avaliação da LP

Aplicabilidade:

- É uma linguagem de uso geral que pode ser usada para resolver diversos problemas desde aplicações web até computação científica. Existe uma grande variedade de módulos adicionais.

Confiabilidade:

- A linguagem é tão confiável quanto Java. Por exemplo: fiscalizando memória(garbage collector) e incentivando o tratamento de exceções.

Facilidade de aprendizado:

- Possui, de maneira simples, grande capacidade de abstração e conjunto reduzido de estruturas de controle. Essas características tornam a linguagem mais fácil de se aprender que C, C++ e Java.

Eficiência:

- Pelo fato de tudo ser objeto, e por existir fiscalização de tipos de dados e memória em tempo de execução, Python é menos eficiente que C e C++ e equiparando-se a Java.

Avaliação da LP

Portabilidade:

- Como Python é escrito em ANSI C portável, compila e executa em praticamente todas as principais plataformas.

Método de projeto:

- Python usa orientação a objetos, e permite o uso dos conceitos do paradigma estruturado separado dos de OO.

Reusabilidade:

- Permite o reuso de funções e classes existentes em módulos e pacotes.
- Por ser orientada a objetos, seu polimorfismo universal também auxilia na criação de código reutilizável e *frameworks*.

Integração:

- Existem o Cpython e Jpython que facilitam a escrita e a importação de códigos e bibliotecas em C e Java, respectivamente, para a linguagem Python.

Avaliação da LP

Custo:

- Por ser uma linguagem de fácil aprendizado, o custo para treinamento e escrita de códigos em Python é baixo;
- Manutenção de código em Python é mais barata, pois os códigos são normalmente muito legíveis.

Escopo:

- Requer a definição explícita de entidades, associada a um escopo de visibilidade.

Expressões e comandos:

- Tem grande variedade de expressões e comandos.

Gerenciamento de memória:

- Disponibiliza um mecanismo de gerência de memória automático que é responsável por alocar e desalocar a memória dos seus objetos.

Avaliação da LP

Passagem de parâmetros:

- Usa mecanismo de passagem de parâmetros por referência;
- Oferece passagem de parâmetros como uma Lista variável.

Encapsulamento e proteção:

- Oferece mecanismos de classes e pacotes;

Verificação de tipos:

- Faz verificação dinâmica de tipos.

Polimorfismo:

- Usa todos os tipos de polimorfismo.

Exceções:

- Apresenta mecanismos embutidos para tratamento de exceções;

Concorrência:

- Oferece threads e recursos para exclusão mútua.

Referências

- www.python.org
- http://www.coremountains.com/products/bytecoat/bytecoat_technical.php
- http://www.digi.com/wiki/developer/index.php/Python_Garbage_Collection
- https://docs.python.org/3/library/gc.html
- https://docs.python.org/3/tutorial/