Programação de Computadores

Instituto de Computação UFF
Departamento de Ciência da Computação

Otton Teixeira da Silveira Filho

Conteúdo

- Mais um tipo numérico
- print() formatado:

  clássico

  pythônico
- Tuplas
- Um exemplo de uso do scipy:

  Fractal de Mandelbrot

Mais sobre tipos número

Python tem a priori vários tipos de dados para representar números. Até agora vimos

- int – Representação finita dos inteiros da matemática
- float – Representação finita do reais da matemática

Python ainda tem como padrão mais um tipo

- complex – uma representação dos números complexos da matemática

complex

Esta discussão só terá utilidade para você caso já tenha visto números complexos

complex

Em Python um complexo é representado como

(re + im j)

onde re é a parte real do complexo, im a parte imaginária e j é raiz quadrada de -1

complex

Para atribuir um valor complexo a um identificador, usaremos a função complex() como abaixo

c = complex(re, im)

onde re é a parte real do número complexo e im é sua parte imaginária

complex

Várias funções já apresentadas no uso de float funcionam para complex como, por exemplo,

- pow()
- abs()

complex

Todos os operadores aritméticos estão sobrecarregados para o uso do tipo complex e temos mais um método útil no uso destes números

- c.conjugate() - devolve o complexo conjugado de c

complex

No entanto, para termos mais recursos, podemos usar vários módulos que dão suporte a este tipo de número

- cmath – um módulo com funções específicas para o uso de funções de argumento complexo
- numpy – vários recursos para números complexos

complex

Vamos a um exemplo onde evocamos vários métodos do módulo cmath

complex

```
# Evocacao do modulo cmath e algumas operacoes com o tipo complex

import cmath as cm

def main():
     c = complex(1.0, 2.0)

     d = c.conjugate()

     i = cm.sqrt(-1.0)

     arcos = cm.acos(1.5)

     cosseno     = cm.cos(c)
     seno        = cm.sin(c)
     exponencial = cm.exp(c)
     polar       = cm.polar(c)

     print("O conjugado de ", c, " e' igual a ", d)
     print("Em coordenadas polares", c, "e' representado como", polar)

     print("O valor da raiz quadrada de -1 e' :", i)
     print("arcos(1.5) = ", arcos)

     print("Para o valor ", c, " os valores para as funcoes abaixo sao")

     print("cos(x) = ", cosseno)
     print("sen(x) = ", seno)
     print("exp(x) = ", exponencial)

main()
```

Abrangência de lista

É comum criarmos listas que são geradas automaticamente que seguem alguma sequência previsível.

Nestes casos podemos usar uma forma sintética de criação de listas.

Ela é denominada em inglês *list comprehension* que traduziremos como **abrangência de lista** e tem similaridades, como veremos, com a definição de funções em matemática

Abrangência de lista

Lembremos da lista que aparece no exemplo de implementação do Crivo de Eratóstenes:

lista cujos os elementos são iguais ao próprio índice. Algo assim

```
inteiros = []
for i in range(n+1):
    inteiros.append(i)
```

que no programa ficou como

Abrangência de lista

Que pode ser gerado por este programa, como já vimos

```
# Geracao de numeros primos usando o crivo de Eratostenes

def main():

    n = int(input("Entre com um numero inteiro positivo : "))

    inteiros = []

# Criacao de elementos da lista com valor igual ao indice
    for i in range(n+1):
        inteiros.append(i)

    print("Inteiros contidos no vetor : ", inteiros)

# Crivo de Eratostenes - sera' atribuido zero aos valores multiplos
# dos anteriores
    for i in range(2, n//2):
        for j in range(2, n//i + 1):
            inteiros[j * i] = 0

    print("Sequencia de primos : ")

    for i in range(2, n+1):
        if inteiros[i] != 0 : print(inteiros[i])

main()
```

Abrangência de lista

No entanto, podemos escrever um programa que gerará esta lista de maneira mais sintética usando abrangência de lista. Algo assim

```
inteiros = [i for i in range(n+1)]
```

No programa fica deste modo

Abrangência de lista

```
# Geracao de numeros primos usando o crivo de Eratostenes
# Exemplo de uso de lista de compreencao

def main():

    n = int(input("Entre com um numero inteiro positivo : "))

# Criacao de elementos da lista com valor igual ao indice

    inteiros = [i for i in range(n+1)]

    print("Inteiros contidos no vetor : ", inteiros)

# Crivo de Eratostenes - sera' atribuido zero aos valores multiplos
# dos anteriores
    for i in range(2, n//2):
        for j in range(2, n//i + 1):
            inteiros[j * i] = 0

    print("Sequencia de primos : ")

    for i in range(2, n+1):
        if inteiros[i] != 0 : print(inteiros[i])

main()
```

Abrangência de lista

Observe novamente no destaque abaixo a maneira que geramos a lista inteiros

```
inteiros = [i for i in range(n+1)]
```

- Não criamos **explicitamente** uma lista vazia
- Geramos o conteúdo da lista de maneira automática e sintética

Abrangência de lista

Este foi um exemplo simples de uma maneira que pode ser útil em muitas situações

No caso que apresentamos a forma é a que se segue

identificador = [expressão for elemento in iterador]

ou seja, você pode fazer operações sobre os elementos gerados pelo for

Abrangência de lista

Mas não se limita a isto. Podemos incluir if para criar condições na lista gerada. Se queremos gerar só uma lista de pares, teríamos

```
inteiros = [i for i in range(n+1) if i %2 == 0]
```

Podemos ainda ter for aninhados e vários if e mesmo if else

Abrangência de lista

Vamos a um exemplo que mostra algumas das possibilidades das listas de compreensão

Abrangência de lista

```
# Exemplos de uso de listas de compreensao
def main():
    valores = [2 * x + 3 for x in range(16)]
    mais_valores = [2 * x + 3 for x in range (16) if x % 2 == 0]
    outros_valores = [ 2 * x + 3 for x in range (16) if x % 2 == 0 if x % 3 == 0]
    x_f = [x for x in range (1, 10)]
    y_f = [x ** 2.0 + 3.0 * x for x in x_f]
    lista = [False if i%2 ==0 else True for i in range(12) ]
    outra_lista = [letra for letra in "tinha uma pedra"]

    print(valores)
    print(mais_valores)
    print(outros_valores)
    for i in range(len(x_f)): print(x_f[i], y_f[i])
    print(lista)
    print(outra_lista)
main()
```

Abrangência de lista

Que tem como saída o abaixo

```
 RESTART: /home/otton/didatico/graduacao/prog/Aulas_prog_2018/Aulas/exercicios_Python/lista_comp1.py
[3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33]
[3, 7, 11, 15, 19, 23, 27, 31]
[3, 15, 27]
1 4.0
2 10.0
3 18.0
4 28.0
5 40.0
6 54.0
7 70.0
8 88.0
9 108.0
[False, True, False, True, False, True, False, True, False, True, False, True]
['t', 'i', 'n', 'h', 'a', ' ', 'u', 'm', 'a', ' ', 'p', 'e', 'd', 'r', 'a']
>>>
```

onde podemos ver os efeitos da estruturação que demos a cada lista.

Abrangência de lista

Há outras possibilidades mas estas são só para apresentar este recurso

- Observe que nem sempre poderemos usar abrangência de lista
- Lembre-se que a legibilidade do código é o mais importante para um programador

Tuplas

Tuplas são um tipo composto de Python que tem muitas semelhanças com as listas.

A principal diferença é que uma tupla é **imutável**

Tuplas

- Uma tupla pode ser construída tendo como elementos qualquer tipo de Python, incluindo outras tuplas
- Podemos acessar cada elemento da tupla por meio de um índice
- Graficamente ela é representada com seus elementos entre parênteses e separados por vírgula, ou seja,

(1, 2, 3, 4) é uma tupla de int

Tuplas

Você deve ter notado que um tipo complex é uma tupla

Tuplas

- Uma tupla com um único elemento deverá ser representada com o elemento seguido de uma vírgula, senão o identificador não será associado a estrutura de uma tupla.

```
tupla = (1, )
```

- Podemos ter tuplas sem elementos
- Os parênteses podem ser dispensados ao definir uma tupla
- Podemos converter um outro tipo em tupla usando tuple()

Tuplas

No exemplo que se segue veremos estas propriedades apresentadas

Tuplas

```
# Exemplo sobre tuplas

def main():

    tupla = (1)
    tupla1 = (1,)
    tupla_int = (1, 2, 3, 4)

    tupla_str = ("Abacate", "Abacaxi", "Ameixa")

    tupla_alternativa = 'a', 'b', 'c'

    tupla_explicita = tuple("abc")

    tupla_nula = ()

    print(tupla, tupla1, tupla_int)

    print(tupla_str, tupla_alternativa, tupla_explicita, tupla_nula)

main()
```

Tuplas

Podemos fazer todas as operações que fazemos com listas, menos as que promovem sua modificação

Tuplas

O essencial na tupla é esta imutabilidade. Isto garante que as informações contidas na tupla não sejam acidentalmente alteradas.

Isto é importante num outro tipo do Python que não descreveremos:

- dicionários

print() formatado

Algumas vezes ficamos com saídas dos programas um tanto confusas

Tais situações podem ser atenuadas pelo controle do formato de saída do print()

Veremos aqui alguns destes recursos de formatação

print() formatado

Apresentaremos as duas versões de formatação, uma mais “clássica“ e outra mais recente, ou como alguns dizem, mais “pythônica“

Primeiro a “clássica“

print() formatado-clássico

Vimos que na tabela ASCII há caracteres não imprimíveis, alguns deles sendo controles de impressão. Apresentaremos apenas dois simbolizados como

- \n – promove um salto para a próxima linha
- \t – promove tabulação na mesma linha

print() formatado-clássico

Temos ainda alguns recursos de formatação, um dos quais já foi apresentado

- end = “<caracter>“ determina o caracter de terminação para o print()
- sep = “<caracter>“ determina o caracter de separação entre os parâmetros impressos

print() formatado-clássico

Podemos ainda determinar o espaço que o parâmetro ocupará na impressão.

print() formatado-clássico

Podemos ainda determinar o espaço que o parâmetro ocupará na impressão. O formato é

%espaço_total.número_de_casas_decimais tipo

ou quando não cabe número de casas decimais

%espaço_total.restrição_de_espaço tipo

print() formatado-clássico

O tipo da formatação é determinado por uma letra. Abaixo apresentamos algumas:

- d – int em decimal
- x – int em hexadecimal
- e – float em notação científica
- f – float em notação decimal
- s – str, ou seja, cadeia de caracteres

print() formatado-clássico

Vamos a um exemplo com números

print() formatado-clássico

```
# Formatacao das saidas do print()
# Numeros

def main():

    i = 2345; j = 3

    x = 3.141592; y = 1.2345

    print("Sem formato (int):")

    print(i, j)

    print("\nFormatado (int):")

    print("%4d %4d" % (i, j))

    print("\nSem formato (float):")

    print(x, y)

    print("\nFormatado (float):")

    print("%10.6f %10.6f" % (x, y))

    print("\nFormatado (float). Mesma variavel com formatos diferentes:")

    print("%14.10f %5.2f" % (x, x))

    print("\nSaltando linha:")

    print("%14.10f \n %5.2f" % (x, x))

main()
```

print() formatado-clássico

Que tem como saída:

```
Sem formato (int):
2345 3

Formatado (int):
2345    3

Sem formato (float):
3.141592 1.2345

Formatado (float):
  3.141592   1.234500

Formatado (float). Mesma variavel com formatos diferentes:
  3.1415920000  3.14

Saltando linha:
  3.1415920000
  3.14
>>>
```

print() formatado-clássico

- Repare que no código introduzi mais uma particularidade do Python:

  podemos atribuir valores a identificadores diferentes na mesma linha, desde que separemos um do outro por ; (ponto e vírgula)

- Repare ainda que temos tuplas no processo de formatação como em

  print("%4d %4d" % (i, j))

- Observe ainda que a especificação do formato se dá numa cadeia de caracteres que deve corresponder aos tipos do que vem na tupla

print() formatado-clássico

Agora um exemplo com listas

print() formatado-clássico

```
# Formatacao das saidas do print()
# Listas

def main():

    lista = [1, 10, 123, 23, 678, 1]
    outra_lista = [34, 23, 1, 456, 89, 12]

    print("Sem formato (lista de int):")

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

    print("\nFormatado:")

    for i in range(len(lista)):
        print("%4d" % lista[i])

    print("\nSem formato (duas listas de int):")

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

    print("\nFormatado (duas listas de int):")

    for i in range(len(lista)):
        print("%5d %5d" % (lista[i], outra_lista[i]))

main()
```

print() formatado-clássico

da qual temos a saída:

```
Sem formato (lista de int):
1
10
123
23
678
1

Formatado:
   1
  10
 123
  23
 678
   1

Sem formato (duas listas de int):
1 34
10 23
123 1
23 456
678 89
1 12

Formatado (duas listas de int):
    1    34
   10    23
  123     1
   23   456
  678    89
    1    12
>>>
```

print() formatado-clássico

Vamos a um exemplo com cadeia de caracteres

print() formatado-clássico

```
# Formatacao das saidas do print()
# Saida de cadeia de caracteres

def main():

    cadeia = "tinha uma pedra"

    print("Sem formatacao:")
    print(cadeia)

    print("\nSem formatacao com impressao de cada elemento:")
    for i in range(len(cadeia)):
        print(cadeia[i])

    print("\nSem formatacao com terminacao nao padrao:")
    for i in range(len(cadeia)):
        print(cadeia[i], end = '-')

    print("\nFormatado com terminacao nao padrao:")
    for i in range(len(cadeia)):
        print("%3s" % cadeia[i], end = ' ')

main()
```

print() formatado-clássico

que tem a saída:

```
Sem formatacao:
tinha uma pedra

Sem formatacao com impressao de cada elemento:
t
i
n
h
a

u
m
a

p
e
d
r
a

Sem formatacao com terminacao nao padrao:
t-i-n-h-a- -u-m-a- -p-e-d-r-a-
Formatado com terminacao nao padrao:
  t   i   n   h   a       u   m   a       p   e   d   r   a
>>>
```

print() formatado-clássico

Repare no que aconteceu na penúltima linha de impressão. Apesar de haver o \n, a linha com

```
print("\nFormatado com terminacao nao padrao:")
```

não aparece o salto devido a termos mudado o padrão do end

print() formatado

Vamos agora para outra maneira de colocarmos formatação no print(), do “modo pythônico de ser“

print() formatado-pythônico

Este modo é mais descritivo que o anterior e o que foi apresentado sobre tipos de formatação aqui também valerá mas teremos controle mais direto

print() formatado-pythônico

Faz uso do método format() cujo os atributos são o que pretende se imprimir com o formato especificado

Coloquemos direto um exemplo com números e depois comentaremos o que o format faz

print() formatado-pythônico

```
...
    # Formatacao de saidas do print
    # Numeros

    def main():

        x = 1.234367; y = 2.345; z = 45.23456; w = 123.678901

        print("Sem formatacao:")

        print("{} {}".format(x, y))
        print("{0} {1}".format(x, y))
        print("{1} {0}".format(x, y))

        print("\nFormatado:")
        print("{0:10.7f} {1:10.7f}".format(x, y))

        print("\nFormato e tabulado:")

        print("{0:10.7f} \t {1:10.7f}".format(x, y))

        print("\nOutro formato e salto de linha:")

        print("{0:5.3f} {1:5.2f}\n".format(x, y))

        print("Numeros formatads e nao formatados:")

        print("{0:5.3f} ## {1:5.2f}".format(x, y), z, w)

        print("\nNumeros formatados com separador nao padrao:")

        print("{0:5.3f} ## {1:5.2f}".format(x, y), z, w, sep = "__")

    main()
```

print() formatado-pythônico

que tem a saída:

```
Sem formatacao:
1.234367 2.345
1.234367 2.345
2.345 1.234367

Formatado:
 1.2343670  2.3450000

Formato e tabulado:
 1.2343670        2.3450000

Outro formato e salto de linha:
1.234  2.35

Numeros formatads e nao formatados:
1.234 ##  2.35 45.23456 123.678901

Numeros formatados com separador nao padrao:
1.234 ##  2.35__45.23456__123.678901
>>>
```

print() formatado-pythônico

Repare que a cadeia de caracteres que determina o formato em uma sequência que determina o formato de cada atributo do método format() e também a ordem na qual ele será impressa.

- Cada atributo do format() será demarcado por {} que conterá as especificidades do formato de cada atributo
- A forma mais elementar é sem nenhum atributo o que é equivalente a não ser formatado
- O numeral colocado entre{} indica a ordem dos atributos no format()
- Segue-se o caracter : e a especificação do formato

print() formatado-pythônico

Há ainda mais recursos e veremos mais alguns deles

print() formatado-pythônico

```
# Formatacao de saidas do print
# Numeros

def main():

    x = 1.234367; y = 2.3456

    print("Sem formatacao:")

    print("{} {}".format(x, y))

    print("\nNa sequencia:")

    print("formatado, ajustado 'a direita, ajustado 'a esquerda, centralizado:")

    print("{0:10.3f} ## {1:10.3f}".format(x, y))

    print("{0:>10.3f} ## {1:>10.3f}".format(x, y))

    print("{0:<10.3f} ## {1:<10.3f}".format(x, y))

    print("{0:^10.3f} ## {1:^10.3f}".format(x, y))

main()
```

print() formatado-pythônico

que tem a saída:

```
Sem formatacao:
1.234367 2.3456

Na sequencia:
formatado, ajustado 'a direita, ajustado 'a esquerda, centralizado:
     1.234 ##      2.346
     1.234 ##      2.346
1.234      ## 2.346
  1.234    ##   2.346
```

print() formatado-pythônico

Temos ainda na sequência do caracter : os caracteres

- < – que ajusta o impresso à esquerda
- > – que ajusta o impresso à direita
- ^ – que ajusta o impresso centralizado

print() formatado-pythônico

Vamos agora ver mais um exemplo com cadeia de caracteres

print() formatado-pythônico

```
# Formatacao de saidas do print
# Cadeia de caracteres

def main():

    frase = "Tinha uma pedra"
    outra_frase = "no meio do caminho"

    print("\nCadeias de caracteres\n")

    print(frase)

    print("\nLimitando o numero de caracteres a dez:")

    print(".........*.")

    print("{0:.10}".format(frase))

    print("\nDuas cadeias de caracteres:")

    print(frase, outra_frase)

    print("\nCada campo com 25 espacos para escrita")

    print("\nNa sequencia:\nsem ajuste, ajustado 'a direita, ajustado 'a esquerda, centralizado:")

    print(".........*.........*.........*.........*.........*.")

    print("{0:25s}{1:25s}".format(frase, outra_frase))

    print("{0:<25s}{1:<25s}".format(frase, outra_frase))

    print("{0:>25s}{1:>25s}".format(frase, outra_frase))

    print("{0:^25s}{1:^25s}".format(frase, outra_frase))

main()
```

print() formatado-pythônico

com a saída dada por:

```
Cadeias de caracteres

Tinha uma pedra

Limitando o numero de caracteres a dez:
.........*.
Tinha uma

Duas cadeias de caracteres:
Tinha uma pedra no meio do caminho

Cada campo com 25 espacos para escrita

Na sequencia:
sem ajuste, ajustado 'a direita, ajustado 'a esquerda, centralizado:
.........*.........*.........*.........*.........*.
Tinha uma pedra          no meio do caminho
Tinha uma pedra          no meio do caminho
          Tinha uma pedra       no meio do caminho
     Tinha uma pedra        no meio do caminho
>>>
```

print() formatado-pythônico

Repare na linha

```
print("{0:.10}".format(frase))
```

- Não especificamos o tipo do atributo do método format(). Neste caso será suposto ser uma cadeia de caracteres
- A sequência de caracteres .10 cria uma restrição de espaço no processo de impressão

print() formatado-pythônico

Existem mais recursos e espero que o apresentado seja um estímulo para que você procure mais

Um exemplo mais elaborado

Apresentaremos um exemplo que usa recursos de vários módulos com a finalidade de gerar uma imagem

Aqui teremos conceitos de computação gráfica, fractais, operações com números complexos

É um exemplo para contemplar mais possibilidades de Python

Fractal de Mandelbrot

Apresentaremos um programa que gera uma versão do Fractal de Mandelbrot

- Aqui apresentamos uma versão modificada da encontrada em

  https://lubosz.wordpress.com/2014/08/16/simple-mandelbrot-set-visualization-in-python-3/

Fractal de Mandelbrot

Este programa evoca uma série de módulos que ainda não usamos:

- numpy - dispõe de operações numéricas e de integração com outras linguagens como FORTRAN, C e C++
- PIL - Python Imaging Library para manipulação de imagens em vários formatos
- colorsys - para conversão entre sistemas de cores

Fractal de Mandelbrot

Ele evoca o ImageMagick que é uma suite de aplicativos para edição não iterativa de imagens

Fractal de Mandelbrot

É um programa que demanda um certo tempo para ser executado. Para diminuir a angústia do usuário, é impresso um contador que indica aproximadamente a porcentagem do processado

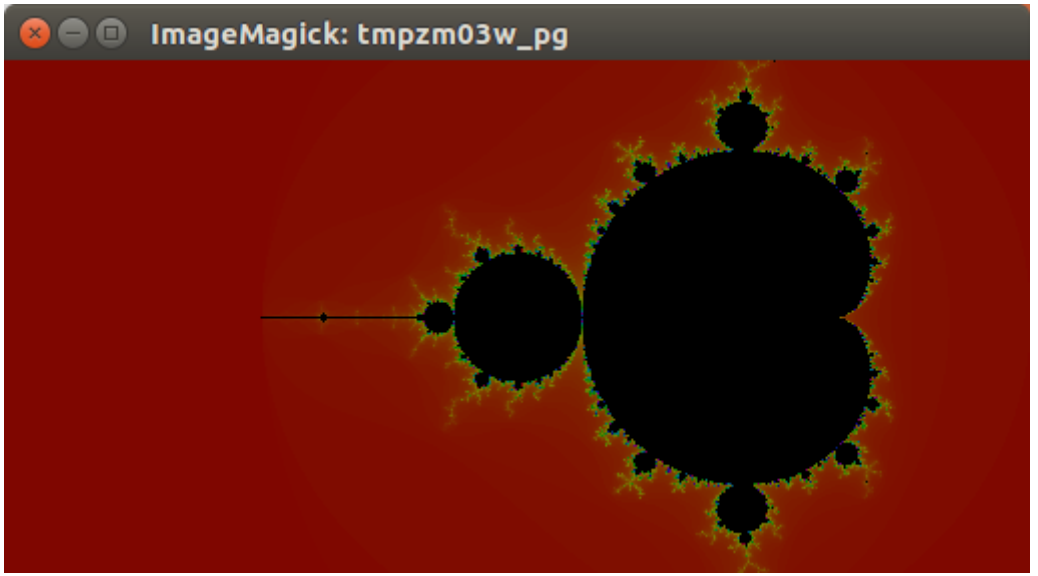
Fractal de Mandelbrot

...

```
# Este programa e' uma variação do programa encontrado em 27/06/2018 na pagina
# https://lubosz.wordpress.com/2014/08/16/
# simple-mandelbrot-set-visualization-in-python-3/

from numpy import complex, array
from PIL import Image
import colorsys

def i_to_rgb(i):
    """ Mapeia cores partindo de valores int """

    cor = 255 * array(colorsys.hsv_to_rgb(i/255.0, 1.0, 0.5))
    return tuple(cor.astype(int))

def mandelbrot(x, y):
    """ Gera uma representacao grafica do Factal de Mandelbrot """

    c0 = complex(x, y); c = 0
    for i in range(1, 1000):
        if abs(c) > 2:
           return i_to_rgb(i)
        c = c * c + c0
    return 0, 0, 0

def main():
    tamanho = 512
    umquarto, tresquartos = tamanho//4, 0.75 * tamanho
    img = Image.new('RGB', (tamanho, int(tamanho/2)))
    pixels = img.load()

    for x in range(img.size[0]):
        print ("%4.2f%%" % (x/tamanho * 100.0))
        for y in range(img.size[1]):
            pixels[x,y] = \
            mandelbrot((x - tresquartos)/umquarto,(y - umquarto)/umquarto)
    img.show()

main()
```

Fractal de Mandelbrot

O final do processamento será gerada a imagem



Fractal de Mandelbrot

Este programa contém uma quantidade grande de informações que já foi apresentada e muitas que envolvem conhecimento de computação gráfica, matemática e dos demais módulos que são usados

Talvez seja um estímulo em direção a mais conhecimentos